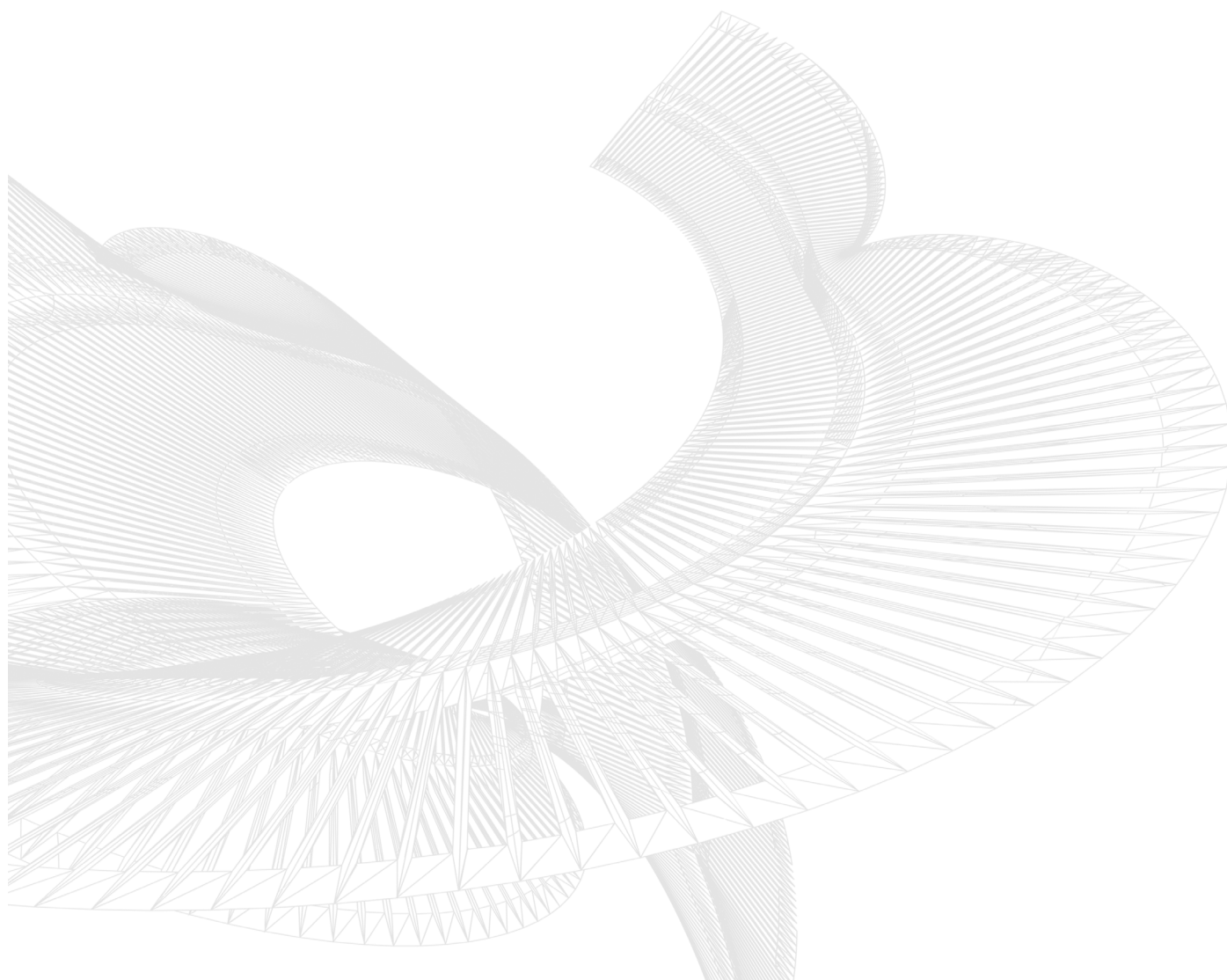


Wolfram *Mathematica* Tutorial Collection

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

©2008 Wolfram Research, Inc.

All rights reserved. No part of this document may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright holder.

Wolfram Research is the holder of the copyright to the Wolfram *Mathematica* software system ("Software") described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, "look and feel," programming language, and compilation of command names. Use of the Software unless pursuant to the terms of a license granted by Wolfram Research or as otherwise authorized by law is an infringement of the copyright.

Wolfram Research, Inc. and Wolfram Media, Inc. ("Wolfram") make no representations, express, statutory, or implied, with respect to the Software (or any aspect thereof), including, without limitation, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Wolfram does not warrant that the functions of the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free. As such, Wolfram does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury or significant loss.

Mathematica, *MathLink*, and *MathSource* are registered trademarks of Wolfram Research, Inc. *J/Link*, *MathLM*, *.NET/Link*, and *webMathematica* are trademarks of Wolfram Research, Inc. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Macintosh is a registered trademark of Apple Computer, Inc. All other trademarks used herein are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc.

Contents

Graphics and Sound

Basic Plotting	1
Options for Graphics	2
Redrawing and Combining Plots	9
Manipulating Options	14
Three-Dimensional Surface Plots	16
Plotting Lists of Data	22
Parametric Plots	25
Some Special Plots	29
Sound	31

The Structure of Graphics and Sound

The Structure of Graphics	33
Two-Dimensional Graphics Elements	39
Graphics Directives and Options	45
Coordinate Systems for Two-Dimensional Graphics	51
Labeling Two-Dimensional Graphics	60
Insetting Objects in Graphics	64
Density and Contour Plots	67
Three-Dimensional Graphics Primitives	71
Three-Dimensional Graphics Directives	76
Coordinate Systems for Three-Dimensional Graphics	81
Lighting and Surface Properties	89
Labeling Three-Dimensional Graphics	96
Efficient Representation of Many Primitives	100
Formats for Text in Graphics	103
Graphics Primitives for Text	106
The Representation of Sound	108
Exporting Graphics and Sounds	110
Importing Graphics and Sounds	113

Editing *Mathematica* Graphics

Introduction	115
Drawing Tools	118
Selecting Graphics Objects	128
Reshaping Graphics Objects	138
Resizing, Cropping, and Adding Margins to Graphics	160
Graphics as Input	163
Interacting with 3D Graphics	164

Graphics and Sound

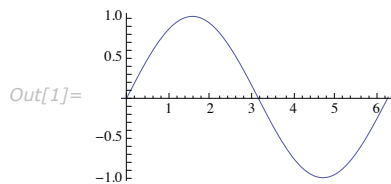
Basic Plotting

<code>Plot[f, {x, x_{min}, x_{max}}]</code>	plot f as a function of x from x_{min} to x_{max}
<code>Plot[f, {x, x₀, x₁, ..., x_k}]</code>	plot f over a series of segments, potentially breaking the curve at each of the x_i
<code>Plot[{f₁, f₂, ...}, {x, x_{min}, x_{max}}]</code>	plot several functions together

Basic plotting functions.

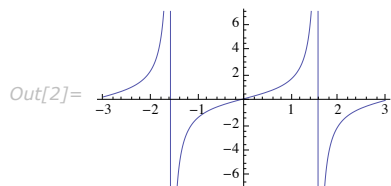
This plots a graph of $\sin(x)$ as a function of x from 0 to 2π .

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



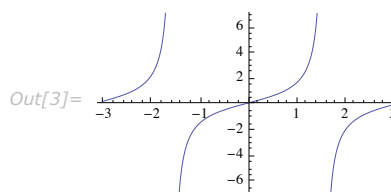
You can plot functions that have singularities. *Mathematica* will try to choose appropriate scales.

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



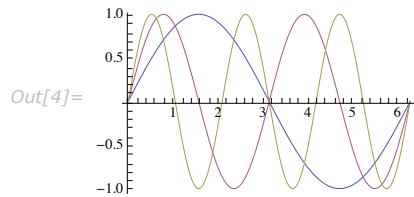
The singularities can be omitted from the plot by specifying them in the plot's range.

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



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}]
```



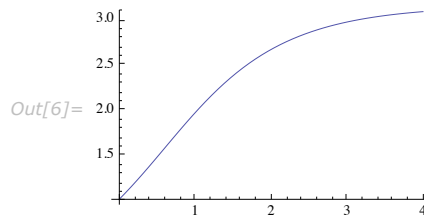
This finds the numerical solution to a differential equation, as discussed in "Numerical Differential Equations".

```
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}]
```



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* $\rightarrow$ *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, xmin, xmax}, 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.

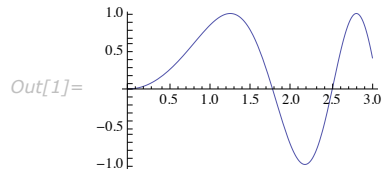
Each time you produce a plot, you can specify options for it. "Redrawing and Combining Plots" will also discuss how you can change some of the options, even after you have produced the plot.

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

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

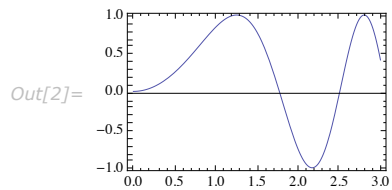
Here is a plot with all options having their default values.

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



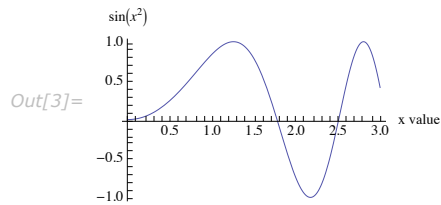
This draws axes on a frame around the plot.

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



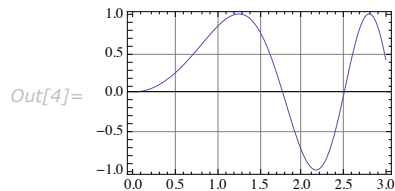
This specifies labels for the x and y axes. The expressions you give as labels are printed just as they would be if they appeared as `TraditionalForm` *Mathematica* output. You can give any piece of text by putting it inside a pair of double quotes.

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



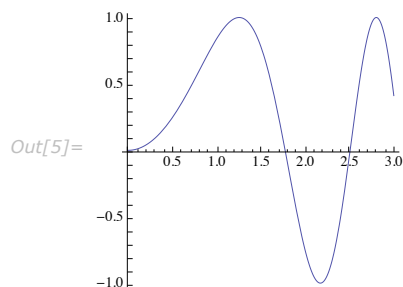
You can give several options at the same time, in any order.

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



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]
```



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.

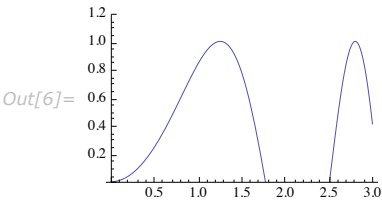
When *Mathematica* makes a plot, it tries to set the x and y scales to include only the “interesting” parts of the plot. If your function increases very rapidly, or has singularities, the parts where it gets too large will be cut off. By specifying the option `PlotRange`, you can control exactly what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

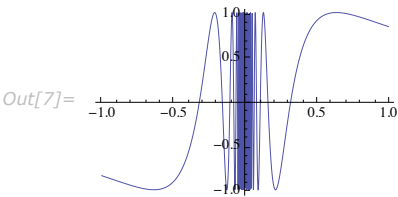
```
In[6]:= Plot[Sin[x^2], {x, 0, 3}, PlotRange -> {0, 1.2}]
```



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}]
```



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.

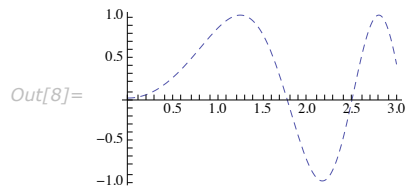
option name	default value	
<code>PlotStyle</code>	Automatic	a list of lists of graphics primitives to use for each curve (see "Graphics Directives and Options")
<code>ClippingStyle</code>	None	what to draw when curves are clipped

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`.

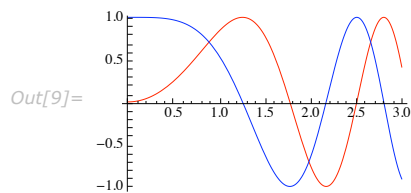
This uses `PlotStyle` to specify a dashed curve.

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



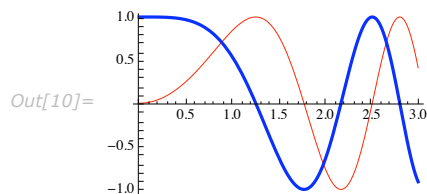
When plotting multiple functions, `PlotStyle` settings in a list are used sequentially for each function.

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



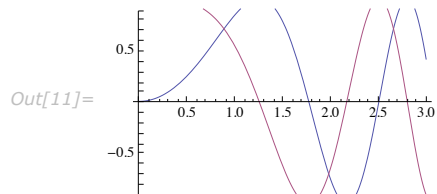
When a `PlotStyle` contains a sublist, the settings are combined.

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



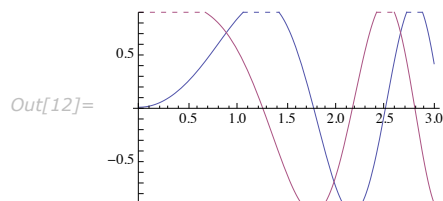
By default nothing is indicated when the `PlotRange` is set, so that it cuts off curves.

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



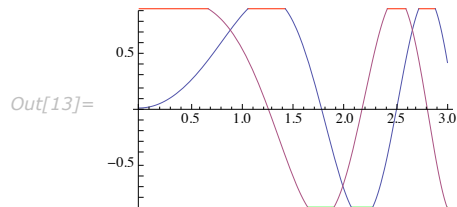
Setting `ClippingStyle` to `Automatic` draws a dashed line where a curve is cut off.

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



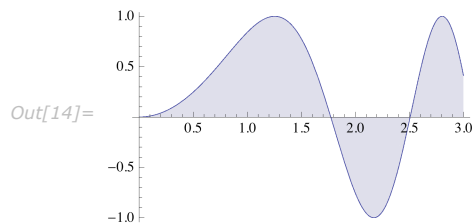
Setting `ClippingStyle` to a list defines the style for the parts cut off at the bottom and top.

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



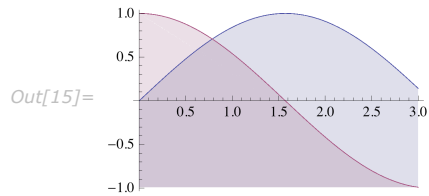
This specifies filling between the curve and the x axis.

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



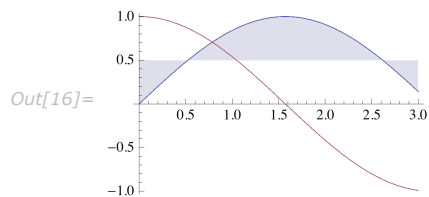
The filling can be specified to extend to an arbitrary height, such as the bottom of the graphic. Filling colors are automatically blended where they overlap.

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



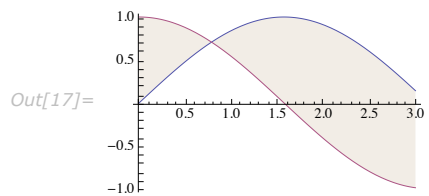
This specifies a specific filling to be used only for the first curve.

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



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]`



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[plot1, plot2, ...]`

combine several plots

`GraphicsGrid[{ {plot1, plot2, ...}, ...}]`

draw an array of plots

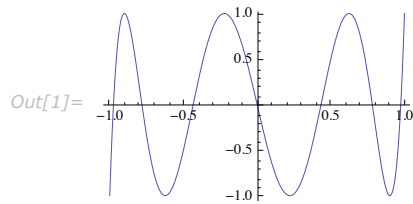
`InputForm[plot]`

show the underlying textual description of the plot

Functions for manipulating plots.

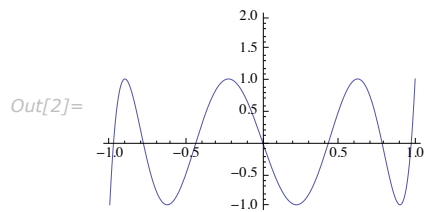
Here is a simple plot.

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



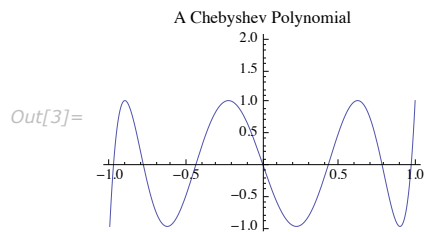
When you redraw the plot, you can change some of the options. This changes the choice of y scale.

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



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

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

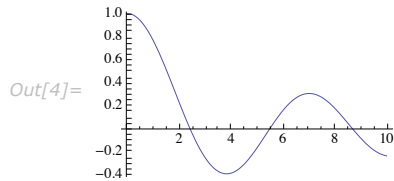


By using `show` with a sequence of different options, you can look at the same plot in many different ways. You may want to do this, for example, if you are trying to find the best possible setting of options.

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.

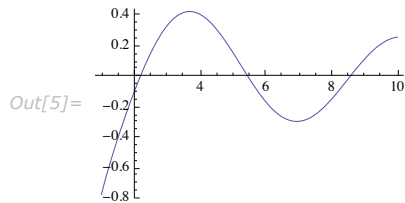
This sets `gj0` to be a plot of $J_0(x)$ from $x = 0$ to 10.

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



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

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



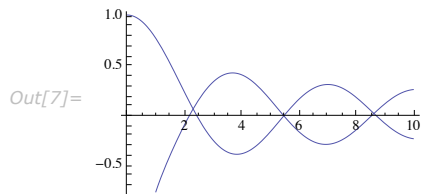
`Plot` specifies an explicit `PlotRange` for each graphic.

```
In[6]:= Options[gj0, PlotRange]
```

```
Out[6]= {PlotRange -> {{0, 10}, {-0.402759, 1.}}}
```

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

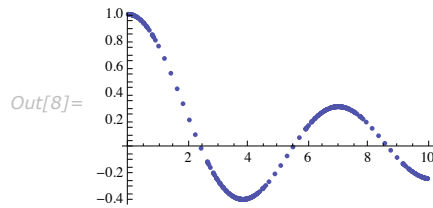
```
In[7]:= gjy = Show[gj0, gy1, PlotRange -> Automatic]
```



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`



Using `show[plot1, plot2, ...]` you can combine several plots into one. `GraphicsGrid` allows you to draw several plots in an array.

`GraphicsGrid[{ {plot11, plot12, ...}, ...}]`

draw a rectangular array of plots

`GraphicsRow[{plot1, plot2, ...}]`

draw several plots side by side

`GraphicsColumn[{plot1, plot2, ...}]`

draw a column of plots

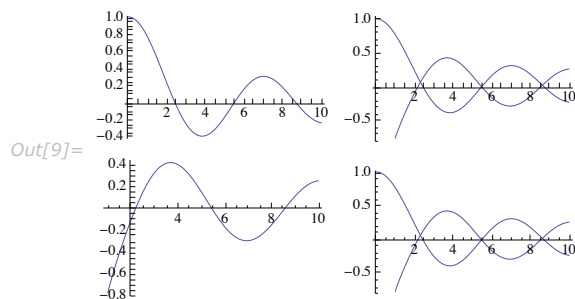
`GraphicsGrid[plots, Spacings -> {h, v}]`

put the specified horizontal and vertical spacing between the plots

Drawing arrays of plots.

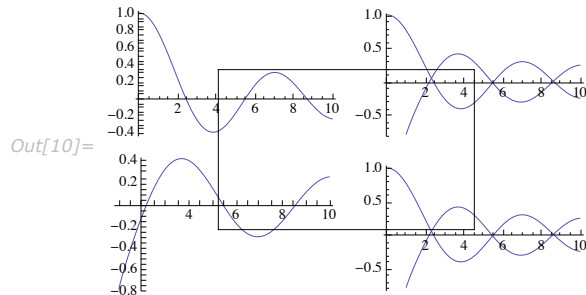
This shows the previous plots in an array.

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



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.

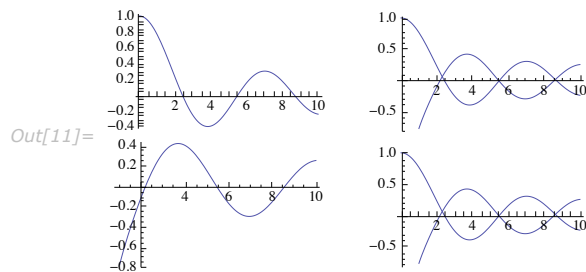
```
In[10]:= Show[%, Frame -> True, FrameTicks -> None]
```



`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.

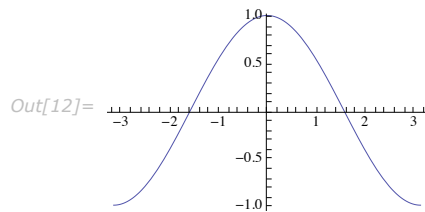
```
In[11]:= GraphicsGrid[{{g10, g11}, {g12, g13}}, Spacings -> {Scaled[.3], Scaled[0]}]
```



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.)

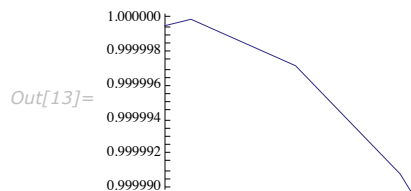
Here is a simple plot.

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



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}}]
```



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.

<code>Options[function]</code>	give a list of the current default settings for all options
<code>Options[function, option]</code>	give the default setting for a particular option
<code>SetOptions[function, option -> value, ...]</code>	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}
```

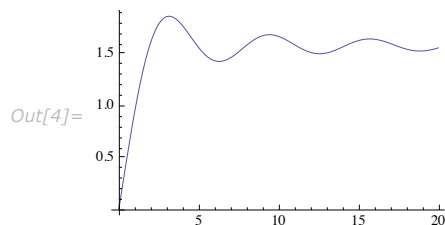
The graphics objects that you get from `Plot` or `Show` store information on the options they use. You can get this information by applying the `Options` function to these graphics objects.

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

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}]
```



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

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

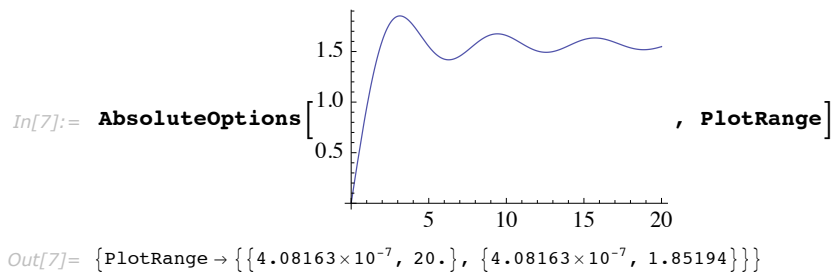
`AbsoluteOptions` gives the *absolute* automatically chosen values used for `PlotRange`.

```
In[6]:= AbsoluteOptions[g, PlotRange]
Out[6]= {PlotRange -> {{4.08163 × 10-7, 20.}, {4.08163 × 10-7, 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.



Three-Dimensional Surface Plots

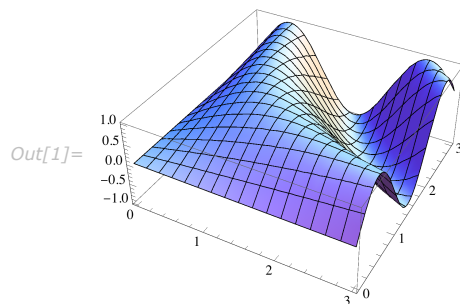
`Plot3D[f, {x, xmin, xmax}, {y, ymin, ymax}]`

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 $\sin(xy)$.

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



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.

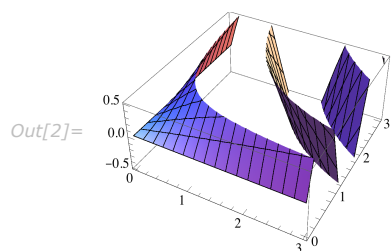
<i>option name</i>	<i>default value</i>	
Axes	True	whether to include axes
AxesLabel	None	labels to be put on the axes: <i>zlabel</i> specifies a label for the <i>z</i> axis, <i>{xlabel, ylabel, zlabel}</i> for all axes
BaseStyle	{}	the default style to use for the plot
Boxed	True	whether to draw a three-dimensional 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 <i>xy</i> mesh should be drawn on the surface
PlotRange	{Full, Full, Automatic}	the range of <i>z</i> 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`.

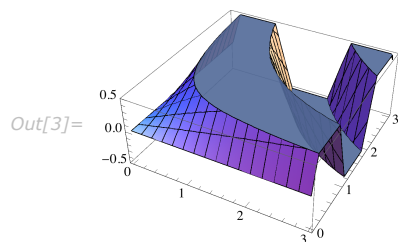
This redraws the previous plot with options changed. With this setting for `PlotRange`, only the part of the surface in the range $-0.5 \leq z \leq 0.5$ is shown.

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



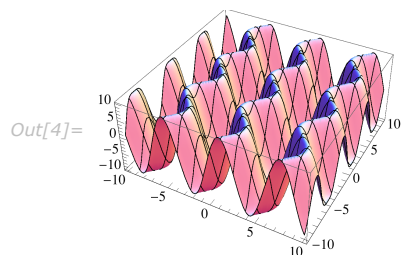
The `ClippingStyle` option of `Plot3D` can be used to fill in the clipped regions.

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



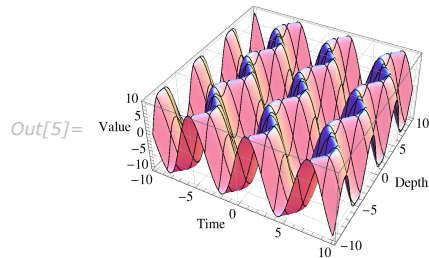
When you make the original plot, you can choose to sample more points. *Mathematica* adaptively samples the plot, adding points for large variations, but occasionally you may still need to specify a greater number of points.

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



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

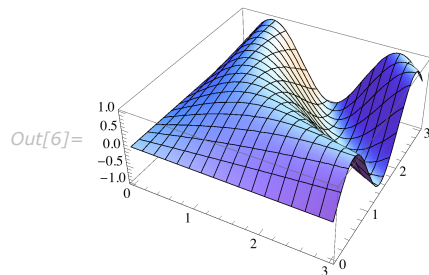
```
In[5]:= Show[%, AxesLabel -> {"Time", "Depth", "Value"}, FaceGrids -> All]
```



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.

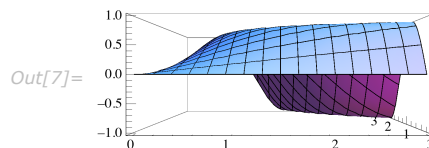
Here is a surface, viewed from the default view point $\{1.3, -2.4, 2\}$. This view point is chosen to be "generic", so that visually confusing coincidental alignments between different parts of your object are unlikely.

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



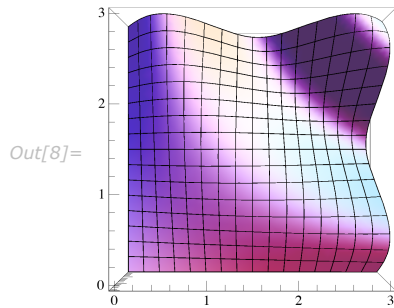
This redraws the picture, with the view point directly in front. Notice the perspective effect that makes the back of the box look much smaller than the front.

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



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

`In[8]:= Show[%, ViewPoint → Above]`



`{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

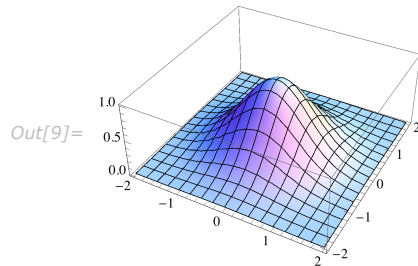
Typical choices for the `ViewPoint` option.

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.

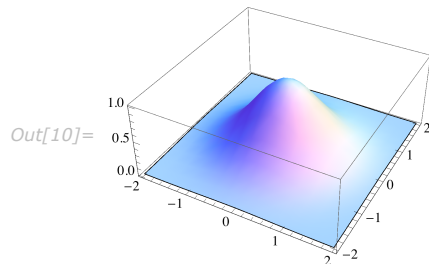
Here is a plot with the default settings for surface rendering options.

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



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.

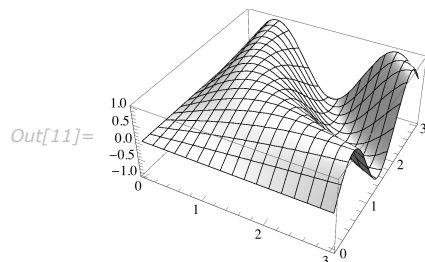
```
In[10]:= Plot3D[Exp[-(x^2 + y^2)], {x, -2, 2}, {y, -2, 2}, Mesh -> None]
```



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.

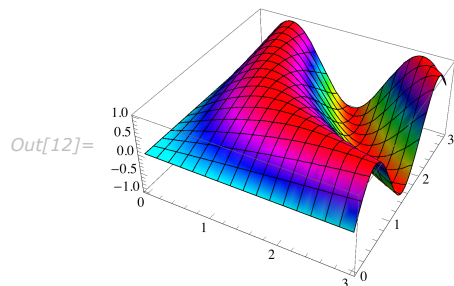
Lighting can also be specified using a string which represents a collection of lighting properties. For example, the option setting `Lighting -> "Neutral"` uses a set of white lights, and so can be faithfully reproduced on a black and white output device such as a printer.

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



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]
```



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.

<code>ListPlot[{y₁, y₂, ...}]</code>	plot $y_1, y_2, \dots$ at x values 1, 2, ...
<code>ListPlot[{ {x₁, y₁}, {x₂, y₂}, ...}]</code>	plot points $(x_1, y_1), \dots$
<code>ListLinePlot[list]</code>	join the points with lines
<code>ListPlot3D[{ {z₁₁, z₁₂, ...}, {z₂₁, z₂₂, ...}, ...}]</code>	make a three-dimensional plot of the array of heights z_{yx}
<code>ListPlot3D[{ {x₁, y₁, z₁}, {x₂, y₂, z₂}, ...}]</code>	make a three-dimensional plot with heights z_i at positions $\{x_i, y_i\}$
<code>ListContourPlot[array]</code>	make a contour plot
<code>ListDensityPlot[array]</code>	make a density plot

Functions for plotting lists of data.

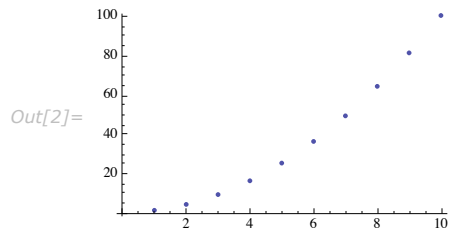
Here is a list of values.

```
In[1]:= t = Table[i^2, {i, 10}]
```

```
Out[1]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

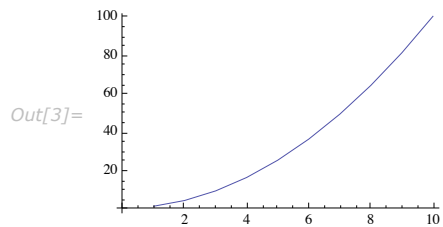
This plots the values.

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



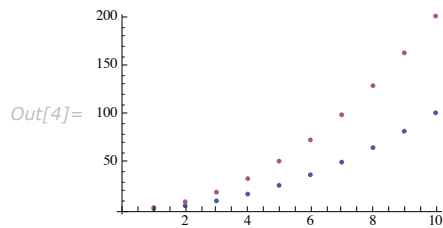
This joins the points with lines.

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



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

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



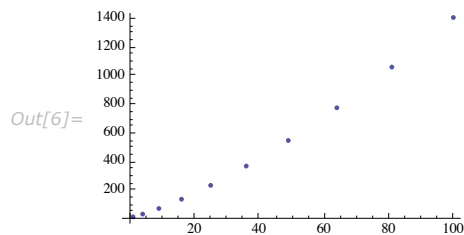
This gives a list of x, y pairs.

`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[%]
```

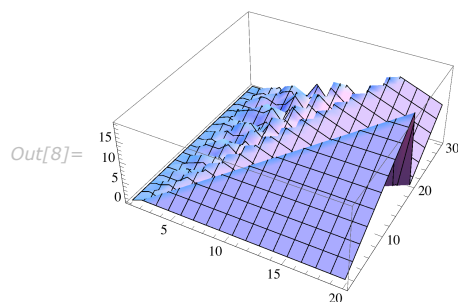


This gives a rectangular array of values. The array is quite large, so we end the input with a semicolon to stop the result from being printed out.

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

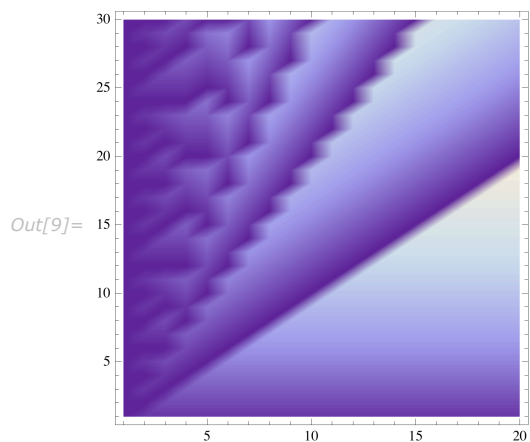
This makes a three-dimensional plot of the array of values.

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



This gives a density plot of the array of values.

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



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[{fx, fy}, {t, tmin, tmax}]
```

make a parametric plot

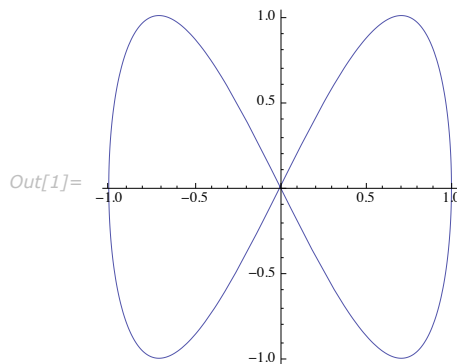
```
ParametricPlot[{ {fx, fy}, {gx, gy}, ... }, {t, tmin, tmax}]
```

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 $\sin[2t]$.

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



```
ParametricPlot3D[{fx, fy, fz}, {t, tmin, tmax}]
```

make a parametric plot of a three-dimensional curve

```
ParametricPlot3D[{ {fx, fy, fz}, {gx, gy, gz}, ... }, {t, tmin, tmax}, {u, umin, umax}]
```

make a parametric plot of a three-dimensional surface

```
ParametricPlot3D[{ {fx, fy, fz}, {gx, gy, gz}, ... }, ...]
```

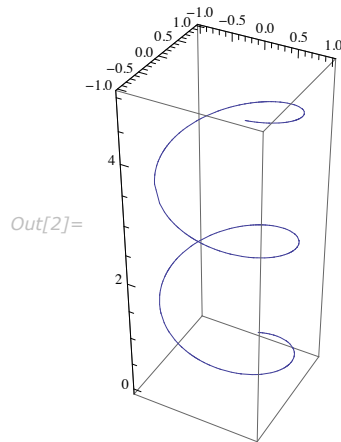
plot several objects together

Three-dimensional parametric plots.

`ParametricPlot3D[{fx, fy, fz}, {t, tmin, tmax}` is the direct analog in three dimensions of `ParametricPlot[{fx, fy}, {t, tmin, tmax}` 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.

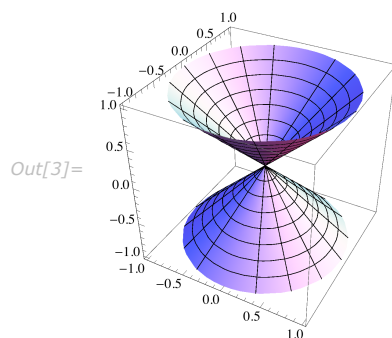
`In[2]:= ParametricPlot3D[{Sin[t], Cos[t], t/3}, {t, 0, 15}]`



`ParametricPlot3D[{fx, fy, fz}, {t, tmin, tmax}, {u, umin, umax}` 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.

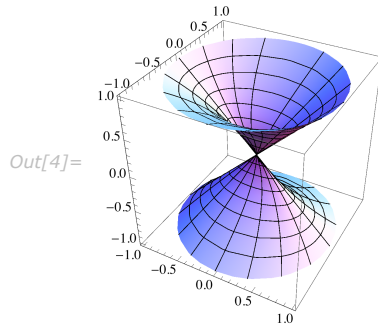
Here the x and y coordinates for the quadrilaterals are given simply by t and u . The result is a surface plot of the kind that can be produced by `Plot3D`.

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



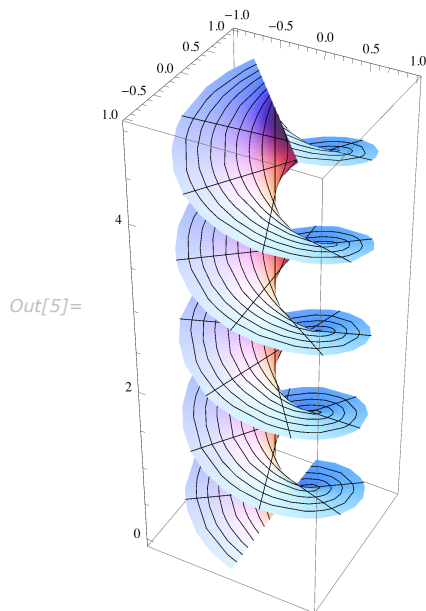
This shows the same surface as before, but with the y coordinates distorted by a quadratic transformation.

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



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}]
```

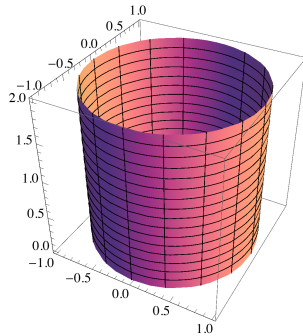


In general, it is possible to construct many complicated surfaces using `ParametricPlot3D`. In each case, you can think of the surfaces as being formed by “distorting” or “rolling up” the t - u coordinate grid in a certain way.

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}]
```

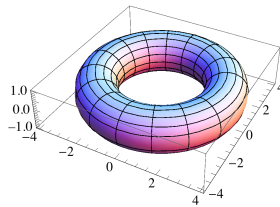
Out[6]=



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}]
```

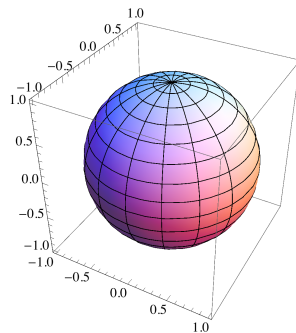
Out[7]=



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

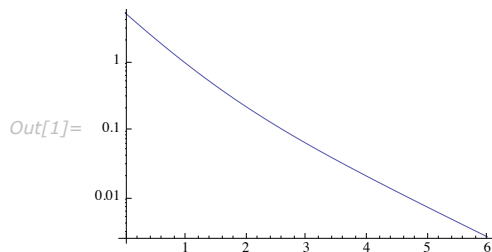
As discussed in "The Structure of Graphics and Sound", *Mathematica* includes a full graphics programming language. In this language, you can set up many different kinds of plots. A few of the common ones are included in standard *Mathematica* packages.

<code>LogPlot[f, {x, x_{min}, x_{max}}]</code>	generate a linear-log plot
<code>LogLinearPlot[f, {x, x_{min}, x_{max}}]</code>	generate a log-linear plot
<code>LogLogPlot[f, {x, x_{min}, x_{max}}]</code>	generate a log-log plot
<code>ListLogPlot[list]</code>	generate a linear-log plot from a list of data
<code>ListLogLinearPlot[list]</code>	generate a log-linear plot from a list of data
<code>ListLogLogPlot[list]</code>	generate a log-log plot from a list of data
<code>PolarPlot[r, {t, t_{min}, t_{max}}]</code>	generate a polar plot of the radius r as a function of angle t
<code>SphericalPlot3D[r, {theta, min, max}, {phi, min, max}]</code>	generate a three-dimensional spherical plot
<code>BarChart[list]</code>	plot a list of data as a bar chart
<code>ErrorListPlot[{ {x₁, y₁, dy₁ }, ... }]</code>	generate a plot with error bars
<code>PieChart[list]</code>	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}]`



Here is a list of the first 10 primes.

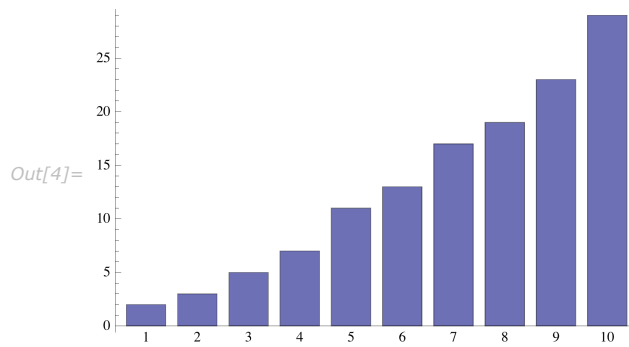
```
In[2]:= p = Table[Prime[n], {n, 10}]
```

```
Out[2]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

Here is a bar chart of the primes.

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

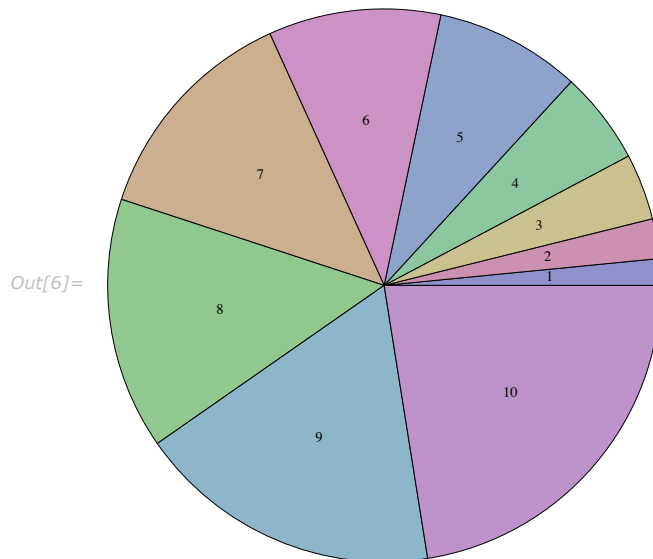
```
In[4]:= BarChart[p]
```



This gives a pie chart.

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

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



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, xmin, xmax}` to plot a function, so also you can use `Play[f, {t, 0, tmax}` 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, tmax}`

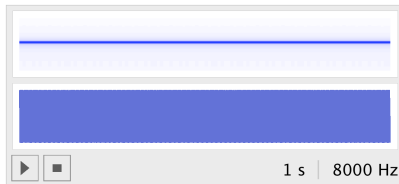
play a sound with amplitude f as a function of time t in seconds

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]=`

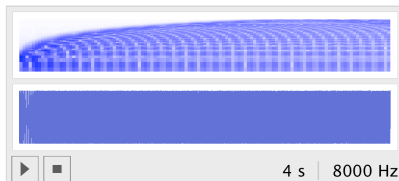


Sounds produced by `Play` can have any waveform. They do not, for example, have to consist of a collection of harmonic pieces. In general, the amplitude function you give to `Play` specifies the instantaneous signal associated with the sound. This signal is typically converted to a voltage, and ultimately 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[{f1, f2, ...}]` to produce stereo sound. In general, *Mathematica* supports any number of sound channels.

```
ListPlay[{a1, a2, ...}, 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".

<code>PlayRange->Automatic</code>	use an internal procedure to scale amplitudes
<code>PlayRange->All</code>	scale so that all amplitudes fit in the allowed range
<code>PlayRange->{a_{min}, a_{max}}</code>	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.

<code>EmitSound[<i>snd</i>]</code>	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

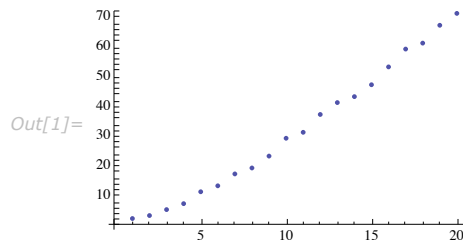
The Structure of Graphics

"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}]]
```



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.

<code>Graphics [list]</code>	general two-dimensional graphics
<code>Graphics3D [list]</code>	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.

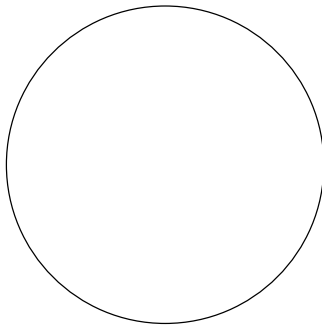
The `Graphics` object is computed by *Mathematica*, but its output is suppressed by the semicolon.

```
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

<code>Show[g, opts]</code>	display a graphics object with new options specified by <i>opts</i>
<code>Show[g₁, g₂, ...]</code>	display several graphics objects combined using the options from <i>g₁</i>
<code>Show[g₁, g₂, ..., opts]</code>	display several graphics objects with new options specified by <i>opts</i>

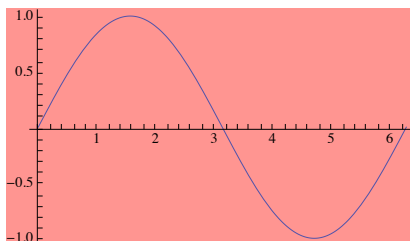
Displaying graphics objects.

`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.

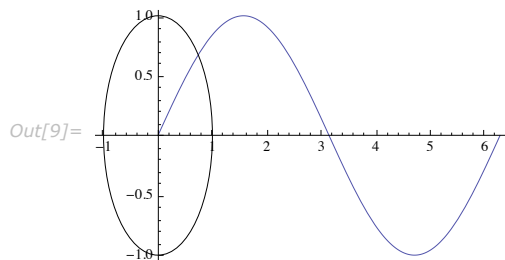
```
In[5]:= g1 = Plot[Sin[x], {x, 0, 2 Pi}];  
Show[g1, Background -> Pink]
```

Out[8]=



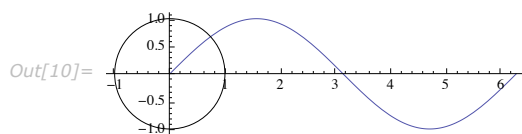
This uses `Show` to combine two graphics. The values used for `PlotRange` and other options are based upon those which were set for the first graphic.

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



Here, new options are specified for the entire graphic.

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



Graphics directives

Examples: `RGBColor`, `Thickness`

Graphics options

Examples: `PlotRange`, `Ticks`, `AspectRatio`,
`ViewPoint`

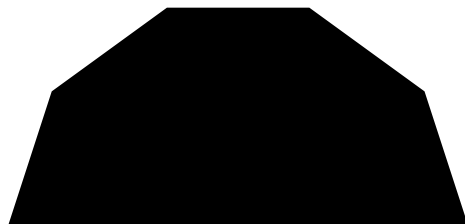
Local and global ways to modify graphics.

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]=



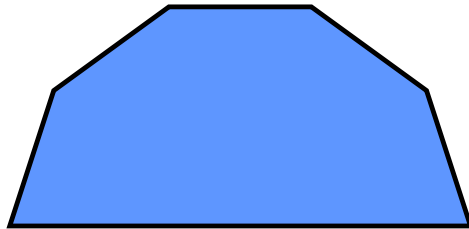
InputForm shows the complete graphics object.

```
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]=

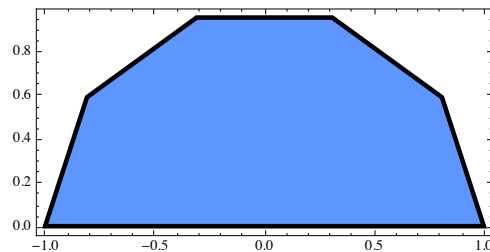


By inserting graphics directives, you can specify how particular graphical elements should be rendered. Often, however, you want to make global modifications to the way a whole graphics object is rendered. You can do this using *graphics options*.

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.

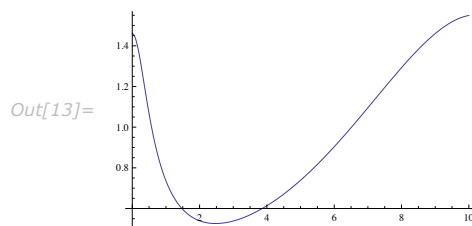
<code>Options[g]</code>	give a list of all graphics options for a graphics object
<code>Options[g, opt]</code>	give the setting for a particular option

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]
```



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}}}
```

<code>FullGraphics[g]</code>	translate objects specified by graphics options into lists of explicit graphics primitives
------------------------------	--

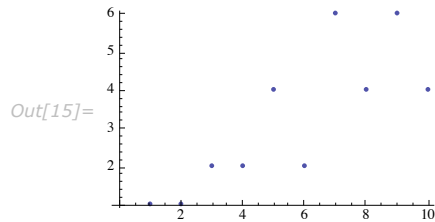
Finding the complete form of a piece of graphics.

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}]]
```



`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

<code>Point[{x,y}]</code>	point at position x, y
<code>Line[{{x₁,y₁},{x₂,y₂},...}]</code>	line through the points $\{x_1, y_1\}, \{x_2, y_2\}, \dots$
<code>Rectangle[{x_{min},y_{min}},{x_{max},y_{max}}]</code>	filled rectangle
<code>Polygon[{{x₁,y₁},{x₂,y₂},...}]</code>	filled polygon with the specified list of corners
<code>Circle[{x,y},r]</code>	circle with radius r centered at x, y
<code>Disk[{x,y},r]</code>	filled disk with radius r centered at x, y
<code>Raster[{{a₁₁,a₁₂,...},{a₂₁,...},...}]</code>	rectangular array of gray levels between 0 and 1
<code>Text[expr,{x,y}]</code>	the text of <i>expr</i> , centered at x, y (see "Graphics Primitives for Text")

Basic two-dimensional graphics elements.

Here is a line primitive.

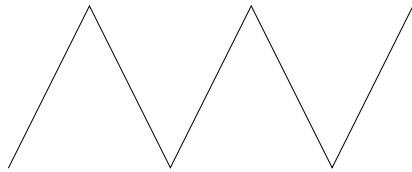
```
In[1]:= sawline = Line[Table[{n, (-1)^n}, {n, 6}]]
```

```
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]
```

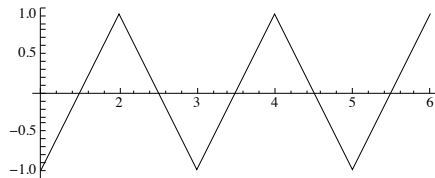
Out[2]=



This redisplay the line, with axes added.

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

Out[3]=

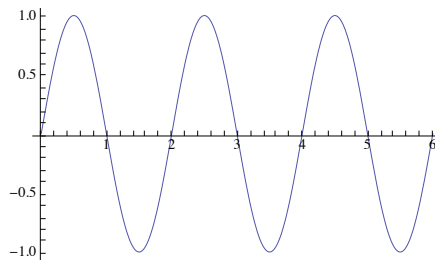


You can combine graphics objects that you have created explicitly from graphics primitives with ones that are produced by functions like `Plot`.

This produces an ordinary *Mathematica* plot.

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

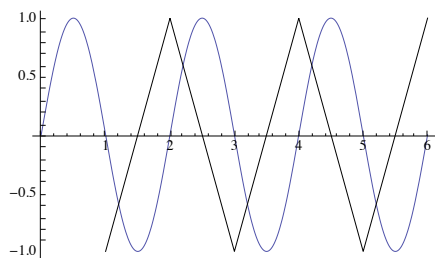
Out[4]=



This combines the plot with the sawtooth picture made above.

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

Out[5]=



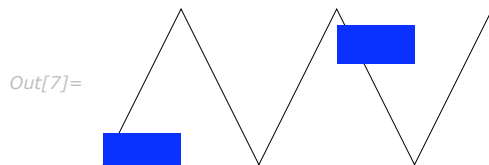
You can combine different graphical elements simply by giving them in a list. In two-dimensional graphics, *Mathematica* will render the elements in exactly the order you give them. Later elements are therefore effectively drawn on top of earlier ones.

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, %}]
```



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.

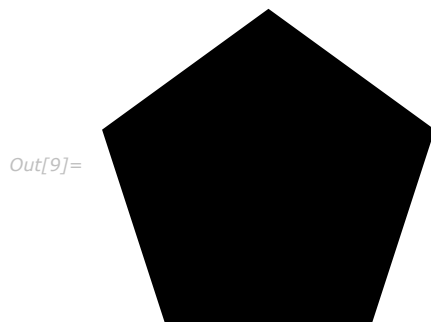
Here are the coordinates of the corners of a regular pentagon.

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

```
Out[8]= {{Sqrt[5/8 + Sqrt[5]/8], 1/4 (-1 + Sqrt[5])}, {Sqrt[5/8 - Sqrt[5]/8], 1/4 (-1 - Sqrt[5])},
{-Sqrt[5/8 - Sqrt[5]/8], 1/4 (-1 + Sqrt[5])}, {-Sqrt[5/8 + Sqrt[5]/8], 1/4 (-1 - Sqrt[5])}, {0, 1}}
```

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

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



<code>Point[{pt₁, pt₂, ...}]</code>	a multipoint consisting of points at $pt_1, pt_2, \dots$
<code>Line[{line₁, line₂, ...}]</code>	a multiline consisting of lines $line_1, line_2, \dots$
<code>Polygon[{poly₁, poly₂, ...}]</code>	a multipolygon consisting of polygons $poly_1, poly_2, \dots$

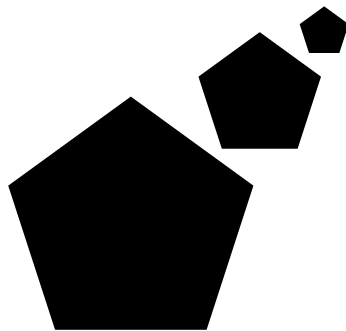
Primitives which can take multiple elements.

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}]]
```

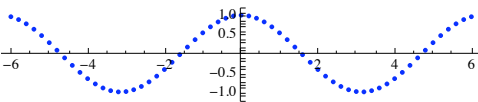
Out[10]=



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]=

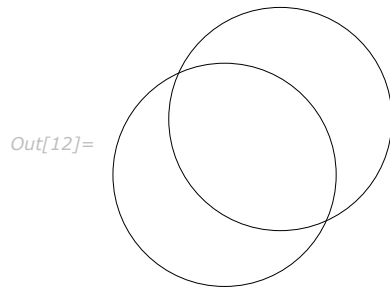


<code>Circle[{x, y}, r]</code>	a circle with radius r centered at the point $\{x, y\}$
<code>Circle[{x, y}, {r_x, r_y}]</code>	an ellipse with semi-axes r_x and r_y
<code>Circle[{x, y}, r, {theta₁, theta₂}]</code>	a circular arc
<code>Circle[{x, y}, {r_x, r_y}, {theta₁, theta₂}]</code>	an elliptical arc
<code>Disk[{x, y}, r], etc.</code>	filled disks

Circles and disks.

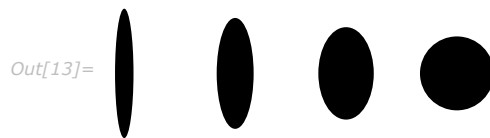
This shows two circles with radius 2.

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



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

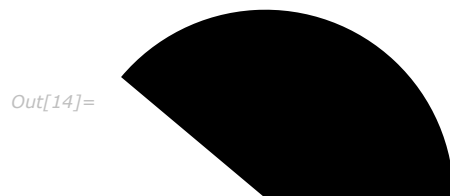
```
In[13]:= Graphics[Table[Disk[{3 n, 0}, {n / 4, 2 - n / 4}], {n, 4}]]
```



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}]]
```



<code>Raster[</code>	
<code>{ {a₁₁, a₁₂, ...}, {a₂₁, ... }, ... }</code>	array of gray levels between 0 and 1
<code>Raster[{ { {a₁₁, o₁₁ }, ... }, ... }</code>	array of gray levels with opacity between 0 and 1

<code>Raster[{{{{r₁₁, g₁₁, b₁₁}}, ...}, ...}]</code>	array of rgb values between 0 and 1
<code>Raster[{{{{r₁₁, g₁₁, b₁₁, o₁₁}}, ...}, ...}]</code>	array of rgb values with opacity between 0 and 1
<code>Raster[array, {{x_{min}, y_{min}}, {x_{max}, y_{max}}}, {z_{min}, z_{max}}]</code>	array of gray levels between z_{min} and z_{max} drawn in the rectangle defined by $\{x_{min}, y_{min}\}$ and $\{x_{max}, y_{max}\}$

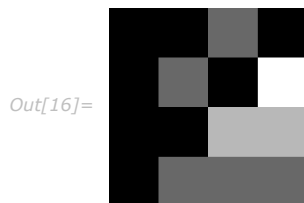
Raster-based graphics elements.

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]]
```



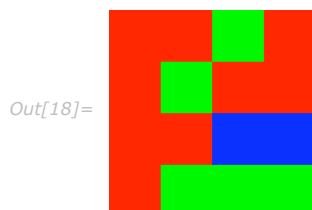
This shows two overlapping copies of the raster.

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



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

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



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.

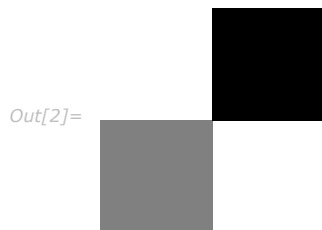
In general, the graphical elements in a particular graphics object can be given in a collection of nested lists. When you insert graphics directives in this kind of structure, the rule is that a particular graphics directive affects all subsequent elements of the list it is in, together with all elements 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[%]
```



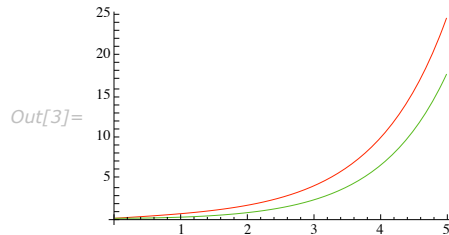
<code>GrayLevel[i]</code>	gray level between 0 (black) and 1 (white)
<code>RGBColor[r,g,b]</code>	color with specified red, green and blue components, each between 0 and 1
<code>Hue[h]</code>	color with hue <i>h</i> between 0 and 1
<code>Hue[h,s,b]</code>	color with specified hue, saturation and brightness,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



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.

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

Graphics directives for points.

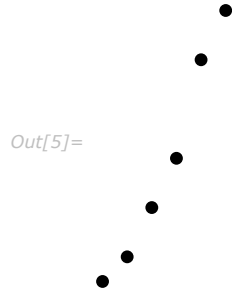
Here is a list of points.

```
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}]}
```

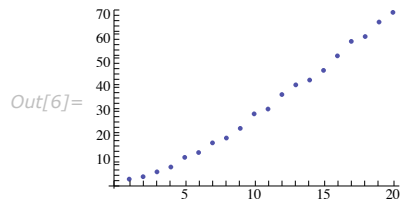
This makes each point have a diameter equal to one-tenth of the width of the plot.

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



Here each point has size 3 in absolute units.

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



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 [$\{w_1, w_2, \dots\}$]	show all lines as a sequence of dashed segments, with lengths $w_1, w_2, \dots$
AbsoluteDashing [$\{w_1, w_2, \dots\}$]	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]=
```

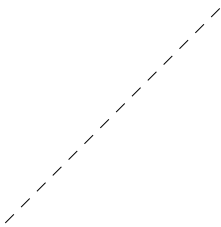


The `Dashing` graphics directive allows you to create lines with various kinds of dashing. The basic idea is to break lines into segments which are alternately drawn and omitted. By changing the lengths of the segments, you can get different line styles. `Dashing` allows you to specify a sequence of 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}}]}
```

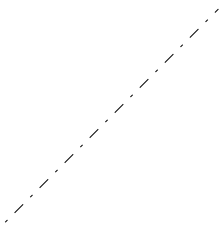
```
Out[9]=
```



This gives a dot-dashed line.

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

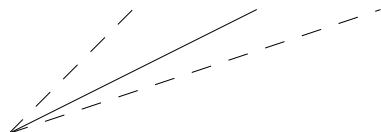
```
Out[10]=
```



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}}]}
```

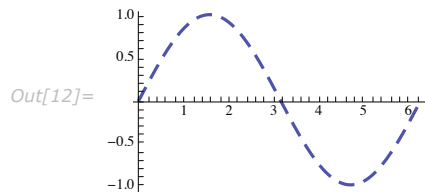
```
Out[11]=
```



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]}}
```



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}}]}]
```



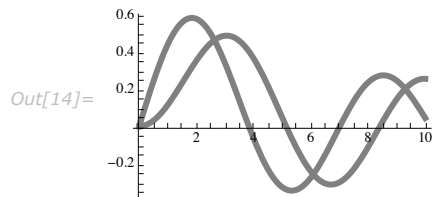
One way to use *Mathematica* graphics directives is to insert them directly into the lists of graphics primitives used by graphics objects. Sometimes, however, you want the graphics directives to be applied more globally, and for example to determine the overall “style” with which a particular type of 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.

<code>PlotStyle->style</code>	specify a style to be used for all curves in <code>Plot</code>
<code>PlotStyle->{{style₁}, {style₂}, ...}</code>	specify styles to be used (cyclically) for a sequence of curves in <code>Plot</code>
<code>MeshStyle->style</code>	specify a style to be used for a mesh in density and surface graphics
<code>BoxStyle->style</code>	specify a style to be used for the bounding box in three-dimensional graphics

Some graphics options for specifying styles.

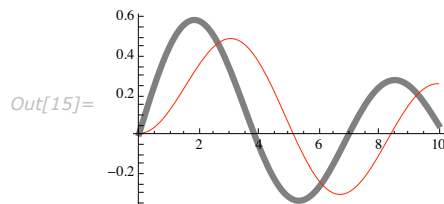
This generates a plot in which all curves are specified to use the same style.

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



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}}]
```



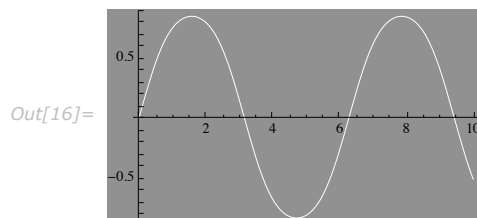
The various “style options” allow you to specify how particular graphical elements in a plot should be rendered. *Mathematica* also provides options that affect the rendering of the whole plot.

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.

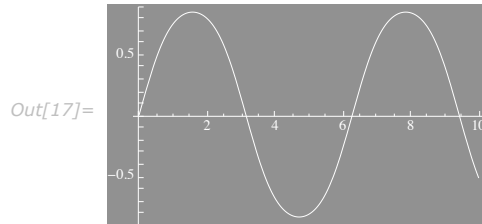
This draws the plot in white on a gray background.

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



This makes the axes white as well.

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



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.

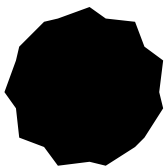
This sets up a polygonal object whose corners have coordinates between roughly ± 1 .

```
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]
```

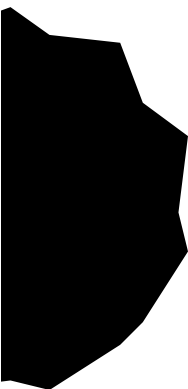
```
Out[2]=
```



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.

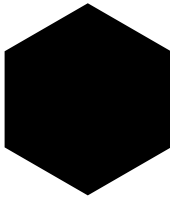
It is important to note that the setting of `AspectRatio` does not affect the meaning of the scaled or display coordinates. These coordinates always run from 0 to 1 across the display area. What `AspectRatio` does is to change the shape of this display area.

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]=



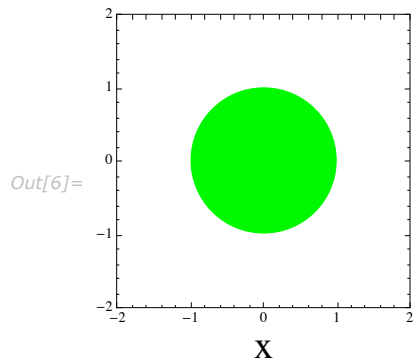
Sometimes, you may find it convenient to specify the display coordinates for a graphical element directly. You can do this by using scaled coordinates `Scaled[{sx, sy}]` rather than `{x, y}`. The scaled coordinates are defined to run from 0 to 1 in x and y , with the origin taken to be at the lower-left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

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

Coordinate systems for two-dimensional graphics.

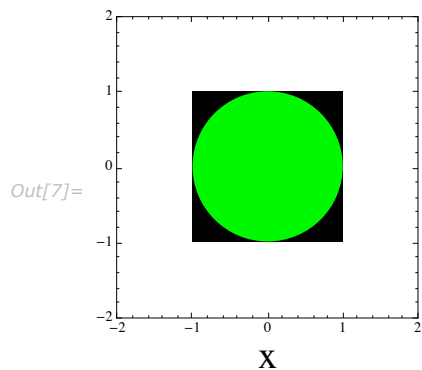
The display area is significantly larger than the plot range due to the frame label.

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



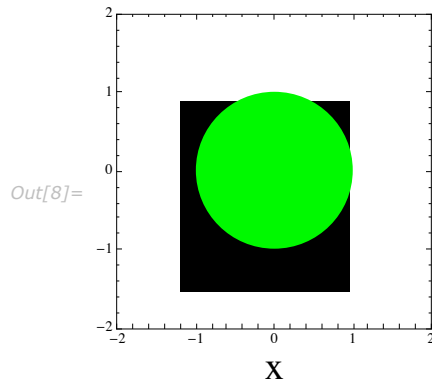
Using `Scaled` coordinates, the rectangle falls at the origin, which is at the center of the specified plot range.

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



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}]]}]
```



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.

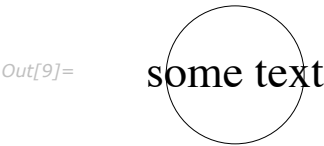
You can use `Scaled[{dsx, dsy}, {x, y}]` to specify a position using a mixture of original and scaled coordinates. In this case, `{x, y}` gives a position in original coordinates, and `{dsx, dsy}` gives the offset from the position in scaled coordinates.

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

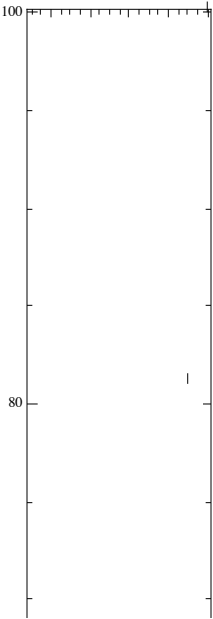


<code>Scaled[{<i>sdx</i>,<i>sd</i><i>y</i>},{<i>x</i>,<i>y</i>}]</code>	scaled offset from original coordinates
<code>ImageScaled[{<i>sdx</i>,<i>sd</i><i>y</i>},{<i>x</i>,<i>y</i>}]</code>	image scaled offset from original coordinates
<code>Offset[{<i>adx</i>,<i>ady</i>},{<i>x</i>,<i>y</i>}]</code>	absolute offset from original coordinates
<code>Offset[{<i>adx</i>,<i>ady</i>},Scaled[{<i>sx</i>,<i>sy</i>}]]</code>	absolute offset from scaled coordinates
<code>Offset[{<i>adx</i>,<i>ady</i>}, ImageScaled[{<i>sx</i>,<i>sy</i>}]]</code>	absolute offset from image scaled coordinates

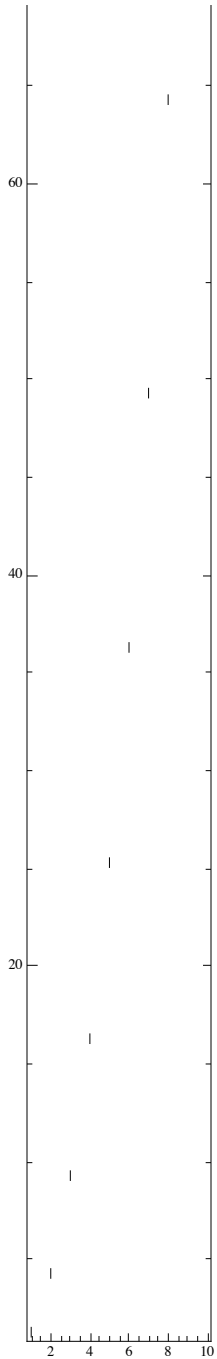
Positions specified as offsets.

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]
```

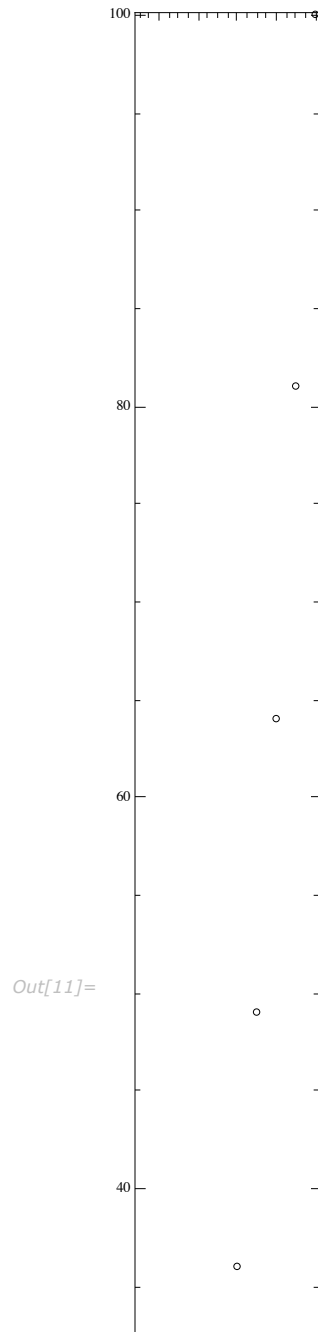


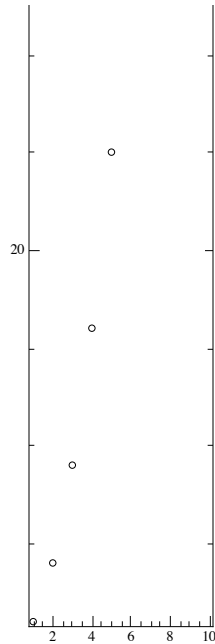
Out[10]=



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]
```





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.)

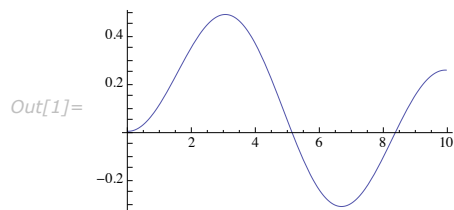
Labeling Two-Dimensional Graphics

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

Ways to label two-dimensional plots.

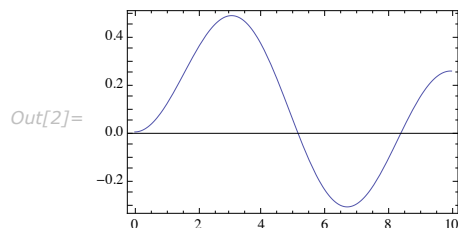
Here is a plot, using the default `Axes -> True`.

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



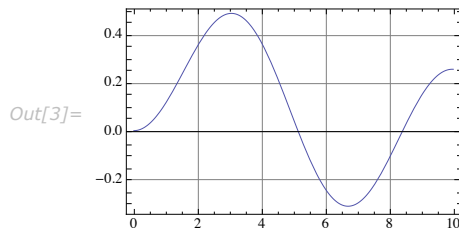
Setting `Frame -> True` generates a frame with axes, and removes tick marks from the ordinary axes.

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



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

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

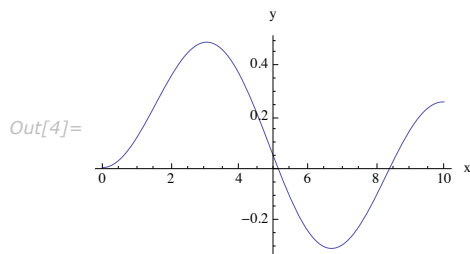


<code>Axes->False</code>	draw no axes
<code>Axes->True</code>	draw both x and y axes
<code>Axes->{False, True}</code>	draw a y axis but no x axis
<code>AxesOrigin->Automatic</code>	choose the crossing point for the axes automatically
<code>AxesOrigin->{x, y}</code>	specify the crossing point
<code>AxesStyle->style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



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

Settings for the `Ticks` option.

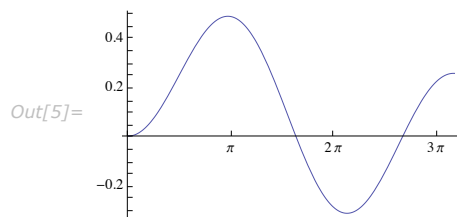
With the default setting `Ticks -> Automatic`, *Mathematica* creates a certain number of major and minor tick marks, and places them on axes at positions which yield the minimum number of decimal digits in the tick labels. In some cases, however, you may want to specify the positions and properties of 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.

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

Tick mark options for each axis.

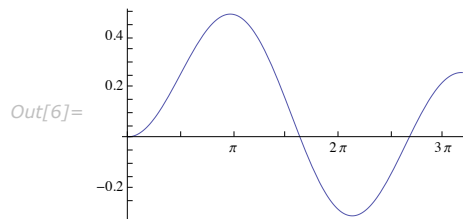
This gives tick marks at specified positions on the x axis, and chooses the tick marks automatically on the y axis.

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



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}]
```



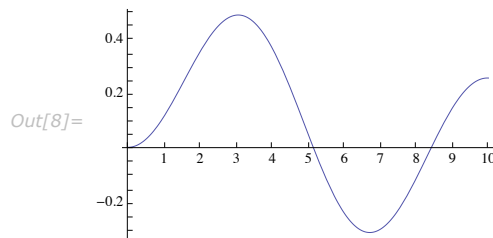
Particularly when you want to create complicated tick mark specifications, it is often convenient to define a "tick mark function" which creates the appropriate tick mark specification given the minimum and maximum values on a particular axis.

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}]
```



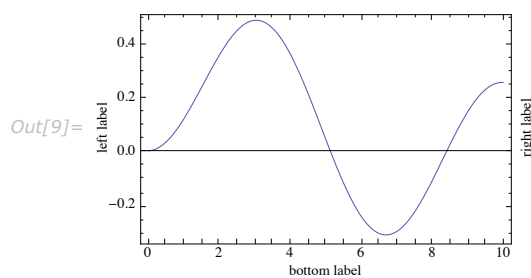
<code>Frame->False</code>	draw no frame
<code>Frame->True</code>	draw a frame around the plot
<code>FrameStyle->style</code>	specify a style for the frame
<code>FrameStyle-></code> <code>{{left,right},{bottom,top}}</code>	specify styles for each edge of the frame
<code>FrameLabel->None</code>	give no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



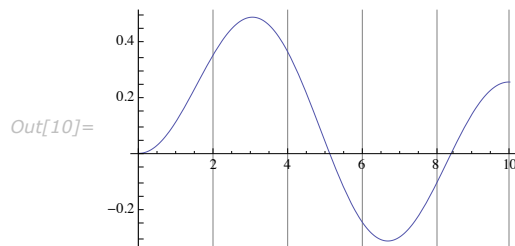
GridLines->None	draw no grid lines
GridLines->Automatic	position grid lines automatically
GridLines->{xgrid,ygrid}	specify grid lines in analogy with tick marks

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}]
```



Insetting Objects in Graphics

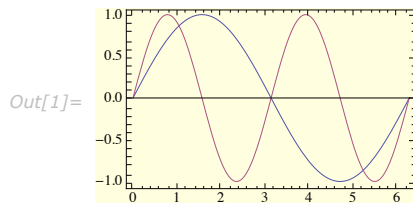
"Redrawing and Combining Plots" describes how you can make regular arrays of plots using `GraphicsGrid`. Using the `Inset` graphics primitive, however, you can combine and superimpose plots in any way.

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

Creating an inset.

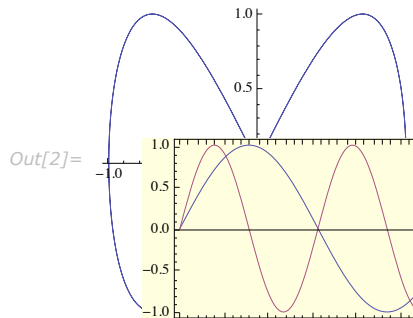
Here is a plot.

```
In[1]:= p1 = Plot[{Sin[x], Sin[2 x]}, {x, 0, 2  $\pi$ },
ImageSize -> 200, Frame -> True, Background -> LightYellow]
```



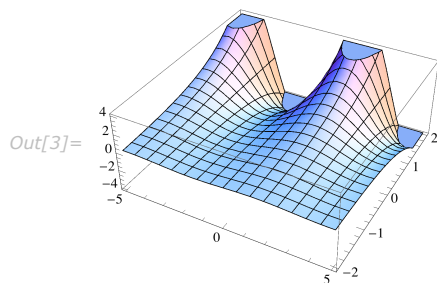
This creates a plot within a parametric plot.

```
In[2]:= ParametricPlot[{Sin[x], Sin[2 x]}, {x, 0, 4  $\pi$ }, Epilog -> Inset[p1, {.3, -.5}]]
```



Here is a three-dimensional plot.

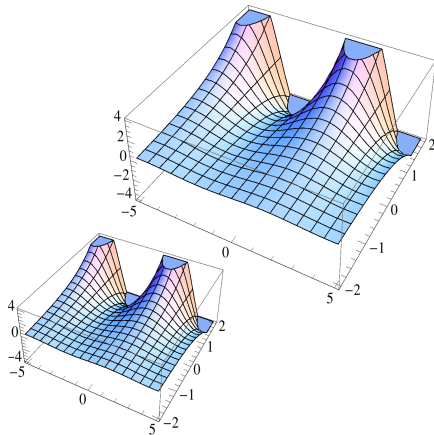
```
In[3]:= p3 = Plot3D[Sin[x] Exp[y], {x, -5, 5}, {y, -2, 2}]
```



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]
```

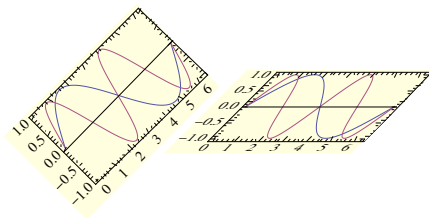
Out[4]=



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, xmin, xmax}, {y, ymin, ymax}]

make a density plot of f

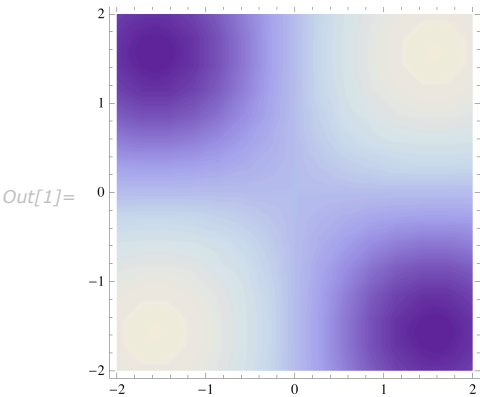
ContourPlot [f, {x, xmin, xmax}, {y, ymin, ymax}]

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}]

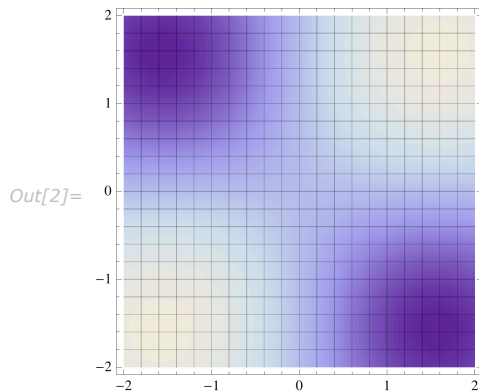


option name	default value	
ColorFunction	Automatic	what colors to use for shading; Hue uses a sequence of hues
Mesh	None	whether to draw a mesh
PlotPoints	Automatic	number of initial sample points in each direction
MaxRecursion	Automatic	the maximum number of recursive subdivision steps to do

Some options for DensityPlot.

You can include a mesh like this.

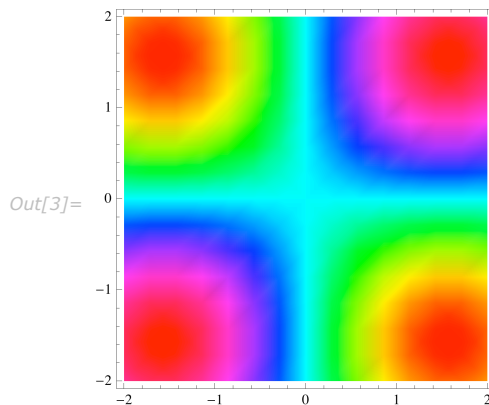
```
In[2]:= DensityPlot[Sin[x] Sin[y], {x, -2, 2}, {y, -2, 2}, Mesh -> 19]
```



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]
```



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`.

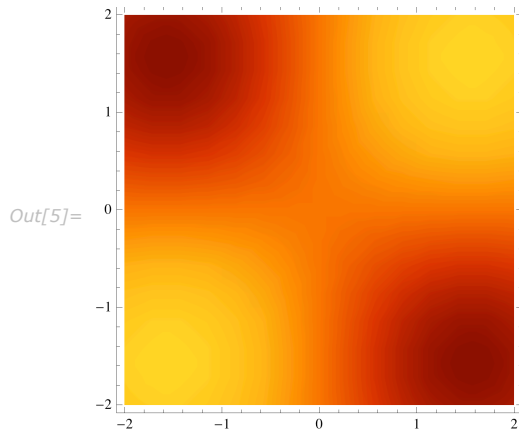
This shows a list of the gradients which can be accessed using `ColorData`.

```
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}
```

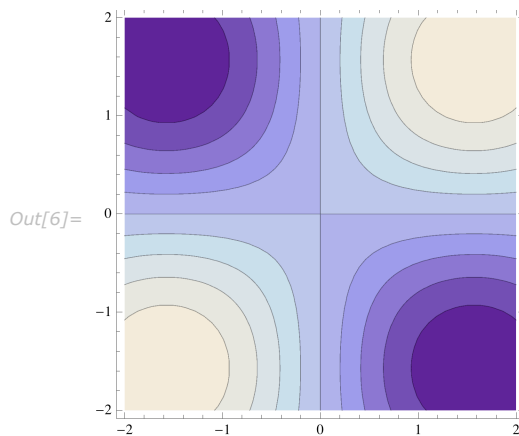
This `DensityPlot` is identical to the one above, but uses the "SolarColors" gradient.

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



This gives a contour plot of the function.

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



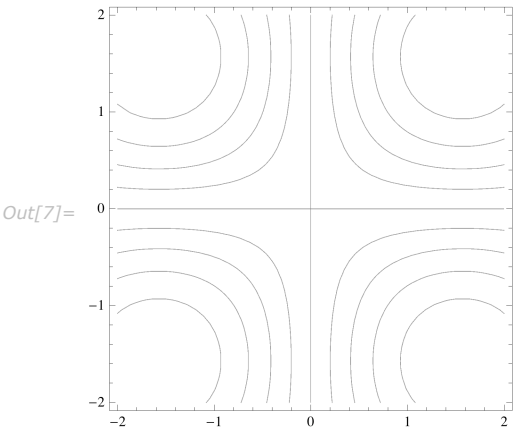
A contour plot gives you essentially a “topographic map” of a function. The contours join points on the surface that have the same height. The default is to have contours corresponding to a sequence of equally spaced z values. Contour plots produced by *Mathematica* are by default shaded, in such a way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

<i>option name</i>	<i>default value</i>	
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 subdivision steps to do

Some options for ContourPlot.

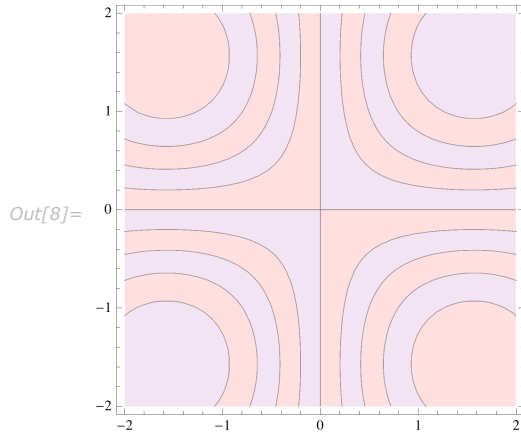
This shows the plot with no shading.

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



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}]
```



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*.

<code>Point[{x,y,z}]</code>	point with coordinates x, y, z
<code>Line[{ {x₁,y₁,z₁}, {x₂,y₂,z₂}, ...}]</code>	line through the points $\{x_1, y_1, z_1\}, \{x_2, y_2, z_2\}, \dots$
<code>Polygon[{ {x₁,y₁,z₁}, {x₂,y₂,z₂}, ...}]</code>	filled polygon with the specified list of corners
<code>Cuboid[{x_{min},y_{min},z_{min}},{x_{max},y_{max},z_{max}}]</code>	cuboid
<code>Text[expr,{x,y,z}]</code>	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.

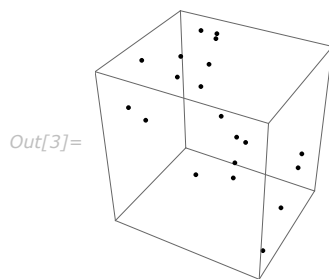
```
In[1]:= rcoord := RandomReal[1., {3}]
```

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

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

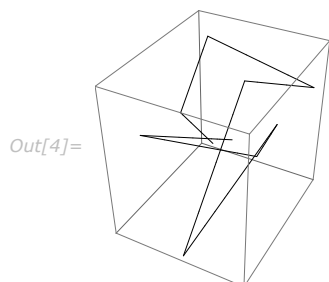
Here is a plot of the points.

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



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

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



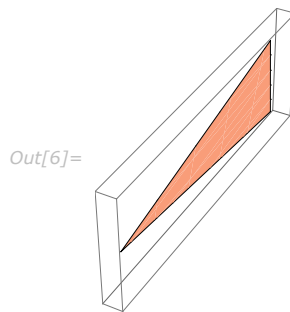
If you give a list of graphics elements in two dimensions, *Mathematica* simply draws each element in turn, with later elements obscuring earlier ones. In three dimensions, however, *Mathematica* collects together all the graphics elements you specify, then displays them as three-dimensional objects,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

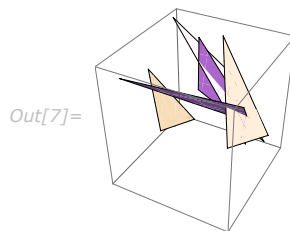
This draws a single random triangle.

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



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

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



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.

<code>Point[{pt₁, pt₂, ...}]</code>	a multipoint consisting of points at $pt_1, pt_2, \dots$
<code>Line[{line₁, line₂, ...}]</code>	a multiline consisting of lines $line_1, line_2, \dots$
<code>Polygon[{poly₁, poly₂, ...}]</code>	a multipolygon consisting of polygons $poly_1, poly_2, \dots$

Primitives which can take multiple elements.

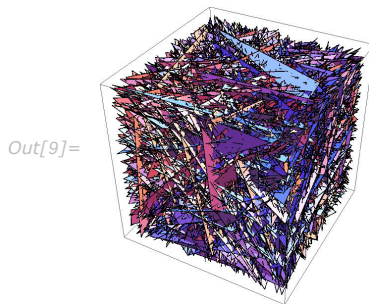
As with the two-dimensional primitives, some three-dimensional graphics primitives have multi-coordinate forms which are a more efficient representation. When dealing with a very large number of primitives, using these multi-coordinate forms where possible can both reduce the memory footprint of the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

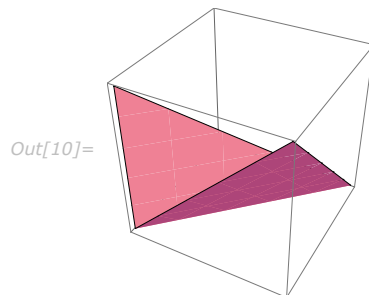
```
In[9]:= Graphics3D[Polygon[Table[rantricoords, {10 000}]]]
```



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.

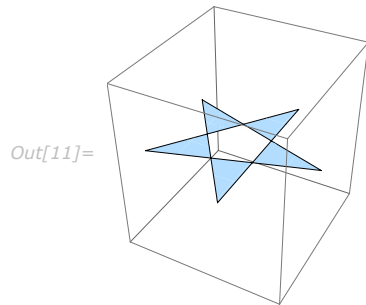
The non-coplanar polygon is broken up into triangles. The interior edge joining the triangles is not outlined like the outer edges of the `Polygon` primitive.

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



Self-intersecting nonconvex polygons are filled according to an even-odd rule that alternates between filling and not filling at each crossing.

```
In[11]:= Graphics3D[Polygon[Table[{Cos[2  $\pi$  k / 5], Sin[2  $\pi$  k / 5], 0}, {k, 0, 8, 2}]]]
```

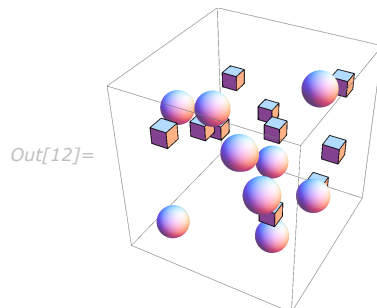


<code>Cuboid[{x,y,z}]</code>	a unit cube with opposite corners having coordinates $\{x, y, z\}$ and $\{x+1, y+1, z+1\}$
<code>Cuboid[{x_{min}, y_{min}, z_{min}}, {x_{max}, y_{max}, z_{max}}]</code>	a cuboid (rectangular parallelepiped) with opposite corners having the specified coordinates
<code>Cylinder[{x₁, y₁, z₁}, {x₂, y₂, z₂}]</code>	a cylinder of radius 1 with endpoints at $\{x_1, y_1, z_1\}$ and $\{x_2, y_2, z_2\}$
<code>Cylinder[{x₁, y₁, z₁}, {x₂, y₂, z₂}, r]</code>	a cylinder of radius r
<code>Sphere[{x,y,z}]</code>	a unit sphere centered at $\{x, y, z\}$
<code>Sphere[{x,y,z}, r]</code>	a sphere of radius r

Cuboid graphics elements.

This draws a number of random unit cubes and spheres in three-dimensional space.

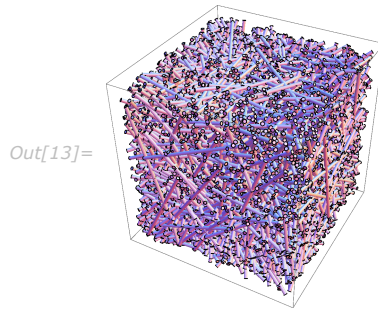
```
In[12]:= Graphics3D[Table[{Cuboid[10 rcoord], Sphere[10 rcoord]}, {10}]]
```



Even though `Cylinder` and `Sphere` produce high-quality renderings, their usage is scalable. A single image can contain thousands of these primitives. When rendering so many primitives, you can increase the efficiency of rendering by using special options to change the number of points used by default to render `Cylinder` and `Sphere`. The `"CylinderPoints"` Method option to `Graphics3D` is used to reduce the rendering quality of each individual cylinder. Sphere quality can be similarly adjusted using `"SpherePoints"`.

Because the cylinders are so small, the number of points used to render them can be reduced with almost no perceptible change.

```
In[13]:= Graphics3D[Table[Cylinder[{rcoord, rcoord}, .01], {10 000}],
  Method → {"CylinderPoints" → 6}]
```



Three-Dimensional Graphics Directives

In three dimensions, just as in two dimensions, you can give various graphics directives to specify how the different elements in a graphics object should be rendered.

All the graphics directives for two dimensions also work in three dimensions. There are however some additional directives in three dimensions.

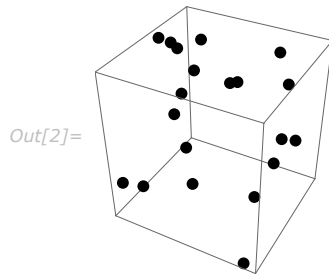
Just as in two dimensions, you can use the directives `PointSize`, `Thickness` and `Dashing` to tell *Mathematica* how to render `Point` and `Line` elements. Note that in three dimensions, the lengths that appear in these directives are measured as fractions of the total width of the display area for your plot.

This generates a list of 20 random points in three dimensions.

```
In[1]:= pts = Table[Point[Table[RandomReal[], {3}]], {20}];
```

This displays the points, with each one being a circle whose diameter is 5% of the display area width.

```
In[2]:= Graphics3D[{PointSize[0.05], pts}]
```



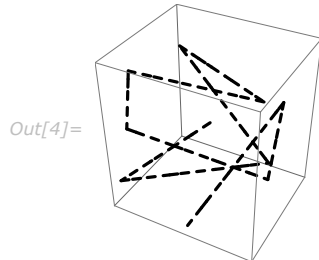
As in two dimensions, you can use `AbsolutePointSize`, `AbsoluteThickness` and `AbsoluteDashing` if you want to measure length in absolute units.

This generates a line through 10 random points in three dimensions.

```
In[3]:= line = Line[Table[RandomReal[], {10}, {3}]];
```

This shows the line dashed, with a thickness of 2 printer's points.

```
In[4]:= Graphics3D[{AbsoluteThickness[2], AbsoluteDashing[{5, 5}], line}]
```



For `Point` and `Line` objects, the color specification directives also work the same in three dimensions as in two dimensions. For `Polygon` objects, however, they can work differently.

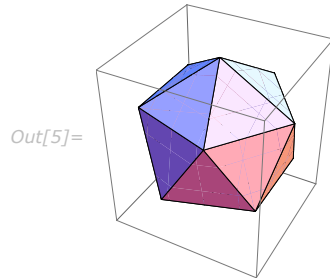
In two dimensions, polygons are always assumed to have an intrinsic color, specified directly by graphics directives such as `RGBColor` and `Opacity`. In three dimensions, however, *Mathematica* generates colors for polygons using a more physical approach based on simulated illumination. Polygons continue to have an intrinsic color defined by color directives, but the final color observed when rendering the graphic may be different based upon the values of the lights shining on the polygon. Polygons are intrinsically white by default.

Lighting->Automatic	use default light placements and values
Lighting->None	disable all lights
Lighting->"Neutral"	light using only white light sources

Some schemes for coloring polygons in three dimensions.

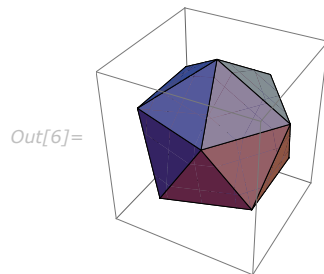
This draws an icosahedron with default lighting. The intrinsic color value of the polygons is white.

```
In[5]:= Graphics3D[{PolyhedronData["Icosahedron", "Faces"]}]
```



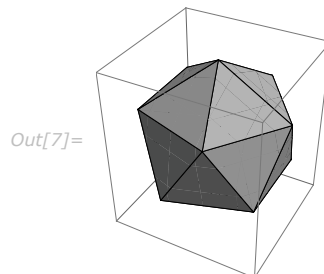
This draws the icosahedron using the same lighting parameters, but defines the intrinsic color value of the polygons to be gray.

```
In[6]:= Graphics3D[{Gray, PolyhedronData["Icosahedron", "Faces"]}]
```



The intrinsic color value of the polygons becomes more obvious when using the "Neutral" lighting scheme.

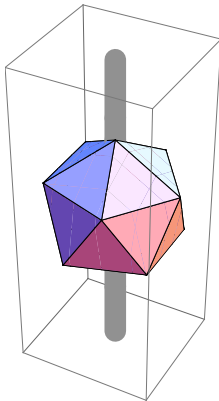
```
In[7]:= Graphics3D[{Gray, PolyhedronData["Icosahedron", "Faces"]}, Lighting -> "Neutral"]
```



This applies the gray color only to the line, which is not affected by the lights.

```
In[8]:= Graphics3D[{PolyhedronData["Icosahedron", "Faces"],
  Gray, Thickness[0.1], Line[{0, 0, -2}, {0, 0, 2}]}]
```

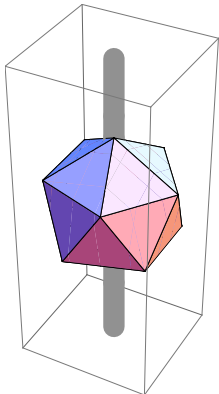
Out[8]=



As with two-dimensional directives, the color directive can be scoped to the line by using a sublist.

```
In[9]:= Graphics3D[{{Gray, Thickness[0.1], Line[{0, 0, -2}, {0, 0, 2}]},
  PolyhedronData["Icosahedron", "Faces"]}]
```

Out[9]=



`EdgeForm[]`

draw no lines at the edges of polygons

`EdgeForm[g]`

use the graphics directives `g` to determine how to draw lines at the edges of polygons

Giving graphics directives for all the edges of polygons.

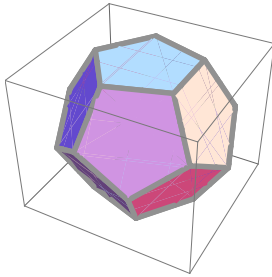
When you render a three-dimensional graphics object in *Mathematica*, there are two kinds of lines that can appear. The first kind are lines from explicit `Line` primitives that you included in the graphics object. The second kind are lines that were generated as the edges of polygons.

You can tell *Mathematica* how to render all lines of the second kind by giving a list of graphics directives inside `EdgeForm`.

This renders a dodecahedron with its edges shown as thick gray lines.

```
In[10]:= Graphics3D[{EdgeForm[{GrayLevel[0.5], Thickness[0.02]}],  
PolyhedronData["Dodecahedron", "Faces"]}]
```

Out[10]=



`FaceForm[gfront, gback]`

use *gfront* graphics directives for the front face of each polygon, and *gback* for the back

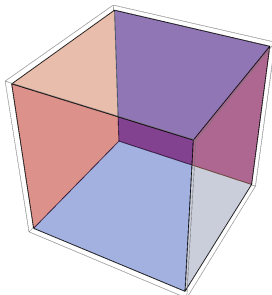
Rendering the fronts and backs of polygons differently.

An important aspect of polygons in three dimensions is that they have both front and back faces. *Mathematica* uses the following convention to define the “front face” of a polygon: if you look at a polygon from the front, then the corners of the polygon will appear counterclockwise, when taken in the order that you specified them.

This makes the front (outside) face of each polygon mostly transparent, and the back (inside) face fully opaque.

```
In[11]:= Graphics3D[{FaceForm[{Opacity[0.3]}, White], PolyhedronData["Cube", "Faces"]}]
```

Out[11]=



Coordinate Systems for Three-Dimensional Graphics

Whenever *Mathematica* draws a three-dimensional object, it always effectively puts a cuboidal box around the object. With the default option setting `Boxed -> True`, *Mathematica* in fact draws the edges of this box explicitly. But in general, *Mathematica* automatically “clips” any parts of your object that extend outside of the cuboidal box.

The option `PlotRange` specifies the range of x , y and z coordinates that *Mathematica* should include in the box. As in two dimensions the default setting is `PlotRange -> Automatic`, which makes *Mathematica* use an internal algorithm to try and include the “interesting parts” of a plot, but drop outlying parts. With `PlotRange -> All`, *Mathematica* will include all parts.

This loads a package defining polyhedron operations.

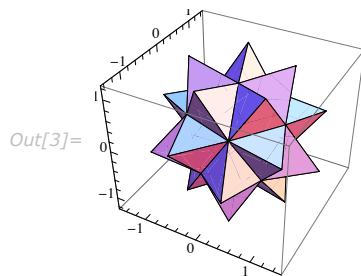
```
In[1]:= << PolyhedronOperations`
```

This creates a stellated icosahedron.

```
In[2]:= stel = Stellate[PolyhedronData["Icosahedron", "Faces"]];
```

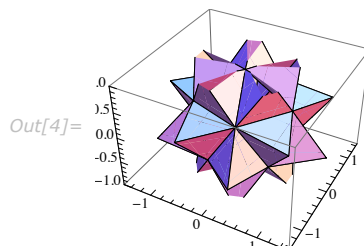
This shows the stellated icosahedron, drawn in a box.

```
In[3]:= Graphics3D[stel, Axes -> True]
```



With this setting for `PlotRange`, many parts of the stellated icosahedron lie outside the box, and are clipped.

```
In[4]:= Show[%, PlotRange -> {-1, 1}]
```



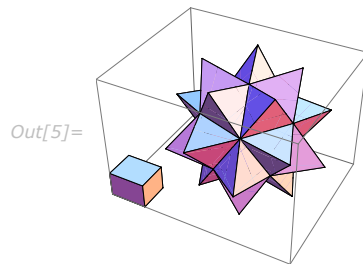
Much as in two dimensions, you can use either “original” or “scaled” coordinates to specify the positions of elements in three-dimensional objects. Scaled coordinates, specified as `Scaled[{sx, sy, sz}]` are taken to run from 0 to 1 in each dimension. The coordinates are set up to define a right-handed coordinate system on the box.

<code>{x, y, z}</code>	original coordinates
<code>Scaled[{sx, sy, sz}]</code>	scaled coordinates, running from 0 to 1 in each dimension

Coordinate systems for three-dimensional objects.

This puts a cuboid in one corner of the box.

```
In[5]:= Graphics3D[{stef, Cuboid[Scaled[{0, 0, 0}], Scaled[{0.2, 0.2, 0.2}]]}]
```



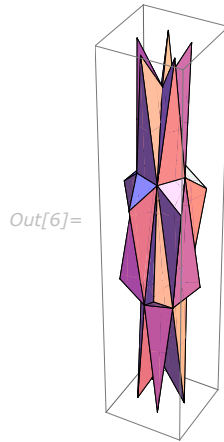
Once you have specified where various graphical elements go inside a three-dimensional box, you must then tell *Mathematica* how to draw the box. The first step is to specify what shape the box should be. This is analogous to specifying the aspect ratio of a two-dimensional plot. In three dimensions, you can use the option `BoxRatios` to specify the ratio of side lengths for the box. For `Graphics3D` objects, the default is `BoxRatios -> Automatic`, specifying that the shape of the box should be determined from the ranges of actual coordinates for its contents.

<code>BoxRatios->{xr, yr, zr}</code>	specify the ratio of side lengths for the box
<code>BoxRatios->Automatic</code>	determine the ratio of side lengths from the range of actual coordinates (default for <code>Graphics3D</code>)

Specifying the shape of the bounding box for three-dimensional objects.

This displays the stellated icosahedron in a tall box.

```
In[6]:= Graphics3D[stel, BoxRatios -> {1, 1, 5}]
```



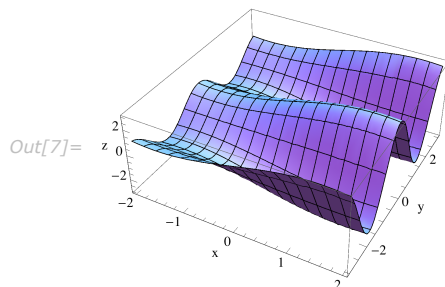
To produce an image of a three-dimensional object, you have to tell *Mathematica* from what view point you want to look at the object. You can do this using the option `ViewPoint`.

Some common settings for this option were given in "Three-Dimensional Surface Plots". In general, however, you can tell *Mathematica* to use any view point.

View points are specified in the form `ViewPoint -> {sx, sy, sz}`. The values si are given in a special coordinate system, in which the center of the box is $\{0, 0, 0\}$. The special coordinates are scaled so that the longest side of the box corresponds to one unit. The lengths of the other sides of the box in this coordinate system are determined by the setting for the `BoxRatios` option. For a cubical box, therefore, each of the special coordinates runs from $-1/2$ to $1/2$ across the box. Note that the view point must always lie outside the box.

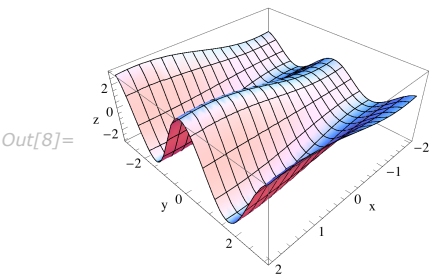
This generates a picture using the default view point $\{1.3, -2.4, 2\}$.

```
In[7]:= surf = Plot3D[(2 + Sin[x]) Cos[2 y],
  {x, -2, 2}, {y, -3, 3}, AxesLabel -> {"x", "y", "z"}]
```



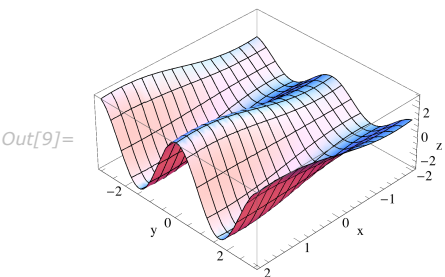
This is what you get with a view point close to one of the corners of the box.

```
In[8]:= Show[surf, ViewPoint -> {1.2, 1.2, 1.2}]
```



As you move away from the box, the perspective effect gets smaller.

```
In[9]:= Show[surf, ViewPoint -> {5, 5, 5}]
```



option name	default value	
ViewPoint	{1.3,-2.4,2}	the point in a special scaled coordinate system from which to view the object
ViewCenter	Automatic	the point in the scaled coordinate system which appears at the center of the final image
ViewVertical	{0,0,1}	the direction in the scaled coordinate system which appears as vertical in the final image
ViewAngle	Automatic	the opening half-angle for a simulated camera used to view the graphic
ViewVector	Automatic	the position and direction of the simulated camera in the graphic's regular coordinate system

Specifying the position and orientation of three-dimensional objects.

In making a picture of a three-dimensional object you have to specify more than just *where* you want to look at the object from. You also have to specify how you want to "frame" the object in your final image. You can do this using the additional options `ViewCenter`, `ViewVertical` and `ViewAngle`.

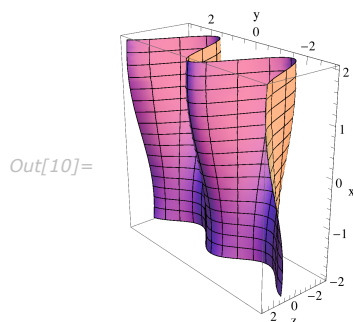
`ViewCenter` allows you to tell *Mathematica* what point in the object should appear at the center of your final image. The point is specified by giving its scaled coordinates, running from 0 to 1 in each direction across the box. With the setting `ViewCenter -> {1/2, 1/2, 1/2}`, the center of the box will therefore appear at the center of your final image. With many choices of view point, however, the box will not appear symmetrical, so this setting for `ViewCenter` will not center the whole box in the final image area. You can do this by setting `ViewCenter -> Automatic`.

`ViewVertical` specifies which way up the object should appear in your final image. The setting for `ViewVertical` gives the direction in scaled coordinates which ends up vertical in the final image. With the default setting `ViewVertical -> {0, 0, 1}`, the z direction in your original coordinate system always ends up vertical in the final image.

Mathematica uses the properties of a simulated camera to visualize the final image. The position, orientation, and facing of the camera are determined by the `ViewCenter`, `ViewVertical`, and `ViewPoint` options. The `ViewAngle` option specifies the width of the opening of the camera lens. The `ViewAngle` specifies, in radians, the maximum angle from the line stretching from the `ViewPoint` to the `ViewCenter` which can be viewed by the camera. The effective viewing angle is double the value of `ViewAngle`. This means that `ViewAngle` can effectively be used to zoom in on a part of the image. The default value of `ViewAngle` resolves to 35° , which is the typical viewing angle for the human eye.

This setting for `ViewVertical` makes the x axis of the box appear vertical in your image.

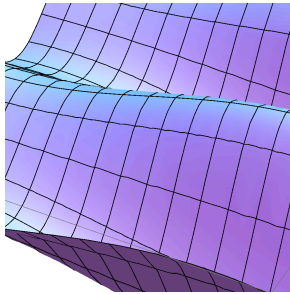
```
In[10]:= Show[surf, ViewVertical -> {1, 0, 0}]
```



This uses `ViewAngle` to effectively zoom in on the center of the image.

```
In[11]:= Show[surf, ViewAngle -> 10 Degree]
```

```
Out[11]=
```



When you set the options `ViewPoint`, `ViewCenter` and `ViewVertical`, you can think about it as specifying how you would look at a physical object. `ViewPoint` specifies where your head is relative to the object. `ViewCenter` specifies where you are looking (the center of your gaze). And `ViewVertical` specifies which way up your head is.

In terms of coordinate systems, settings for `ViewPoint`, `ViewCenter` and `ViewVertical` specify how coordinates in the three-dimensional box should be transformed into coordinates for your image in the final display area.

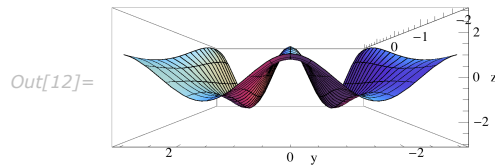
<code>ViewVector -> Automatic</code>	uses the values of the <code>ViewPoint</code> and <code>ViewCenter</code> options to determine the position and facing of the simulated camera
<code>ViewVector -> {x, y, z}</code>	position of the camera in the coordinates used for objects; the facing of the camera is determined by the <code>ViewCenter</code> option
<code>ViewVector -> {{x, y, z}, {tx, ty, tz}}</code>	position of the camera and of the point the camera is focused on in the coordinates used for objects

Possible values of the `ViewVector` option.

The position and facing of the camera can be fully determined by the `ViewPoint` and `ViewCenter` options, but the `ViewVector` option offers a useful generalization. Instead of specifying the position and facing of the camera using scaled coordinates, `ViewVector` provides the ability to position the camera using the same coordinate system used to position objects within the graphic.

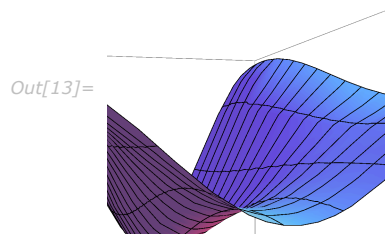
This specifies that the camera should be placed on the negative x axis and facing toward the center of the graphic.

```
In[12]:= Show[surf, ViewVector → {-5, 0, 0}]
```



The camera is in the same position but pointing in a different direction. In combination with `ViewAngle`, this zooms in on a particular section of the graphic.

```
In[13]:= Show[surf, ViewVector → {{-5, 0, 0}, {2, -3, 2}}, ViewAngle → 20 Degree]
```



Once you have obtained a two-dimensional image of a three-dimensional object, there are still some issues about how this image should be rendered. The issues however are identical to those that occur for two-dimensional graphics. Thus, for example, you can modify the final shape of your image by changing the `AspectRatio` option. And you specify what region of your whole display area your image should take up by setting the `PlotRegion` option.

drag	rotate the graphic about its center
Ctrl+drag	zoom into or out of the graphic
Shift+drag	pan across the graphic in the plane of the screen

Mouse gestures used for interacting with three-dimensional graphics.

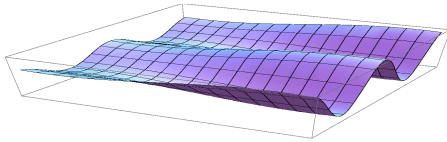
When interactively modifying graphics, *Mathematica* makes changes to the view options. If you have specified the position of the camera using `ViewPoint`, then rotating the graphic causes *Mathematica* to change the value of the `ViewPoint` option. If the position of the camera is specified using `ViewVector`, interactive rotation will instead change the value of that option. In both cases, interactive rotation can also affect the value of the `ViewVertical` option. Interac-

tive zooming of the graphic corresponds directly to changing the `ViewAngle` option. Interactively panning the graphic changes values of the `ViewCenter` option.

This modifies the aspect ratio of the final image.

```
In[14]:= Show[surf, Axes -> False, AspectRatio -> 0.3]
```

Out[14]=

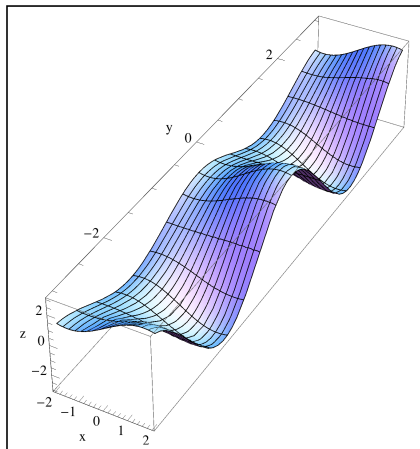


Mathematica usually scales the images of three-dimensional objects to be as large as possible, given the display area you specify. Although in most cases this scaling is what you want, it does have the consequence that the size at which a particular three-dimensional object is drawn may vary with the orientation of the object. You can set the option `SphericalRegion -> True` to avoid such variation. With this option setting, *Mathematica* effectively puts a sphere around the three-dimensional bounding box, and scales the final image so that the whole of this sphere fits inside the display area you specify. The sphere has its center at the center of the bounding box, and is drawn so that the bounding box just fits inside it.

This draws a rather elongated version of the plot.

```
In[15]:= Framed[Show[surf, BoxRatios -> {1, 5, 1}]]
```

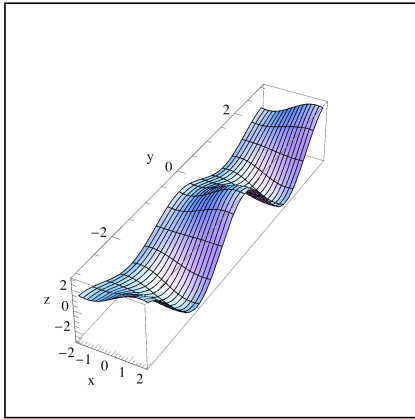
Out[15]=



With `SphericalRegion -> True`, the final image is scaled so that a sphere placed around the bounding box would fit in the display area.

```
In[16]:= Framed[Show[surf, BoxRatios -> {1, 5, 1}, SphericalRegion -> True]]
```

Out[16]=



By setting `SphericalRegion -> True`, you can make the scaling of an object consistent for all orientations of the object. This is useful if you create animated sequences which show a particular object in several different orientations.

`SphericalRegion->False`

scale three-dimensional images to be as large as possible

`SphericalRegion->True`

scale images so that a sphere drawn around the three-dimensional bounding box would fit in the final display area

Changing the magnification of three-dimensional images.

Lighting and Surface Properties

With the default option setting `Lighting -> Automatic`, *Mathematica* uses a simulated lighting model to determine how to color polygons in three-dimensional graphics.

Mathematica allows you to specify various components to the illumination of an object. One component is the "ambient lighting", which produces uniform shading all over the object. Other components are directional, and produce different shading on different parts of the object. "Point lighting" simulates light emanating in all directions from one point in space. "Spot lighting" is similar to point lighting, but emanates a cone of light in a particular direction. "Directional lighting" simulates a uniform field of light pointing in the given direction. *Mathematica* adds together the light from all of these sources in determining the total illumination of a particular polygon.

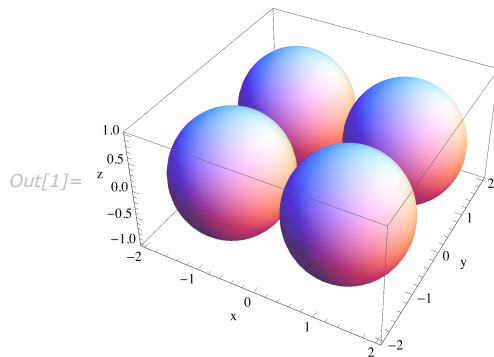
<code>{"Ambient", color}</code>	uniform ambient lighting
<code>{"Directional", color, {pos₁, pos₂}</code>	directional lighting parallel to the vector from pos_1 to pos_2
<code>{"Point", color, pos}</code>	spherical point light source at position pos
<code>{"Spot", color, {pos, tar}, α}</code>	spotlight at position pos aimed at the target position tar with a half-angle opening of α
<code>Lighting -> {light₁, light₂, ...}</code>	a number of lights

Methods for specifying light sources.

The default lighting used by *Mathematica* involves three point light sources, and no ambient component. The light sources are colored respectively red, green and blue, and are placed at 45° angles on the right-hand side of the object.

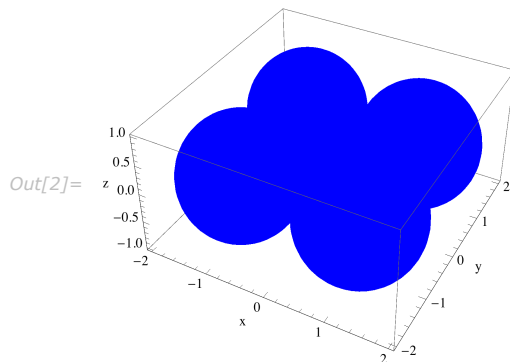
Here is a sphere, shaded using simulated lighting using the default set of lights.

```
In[1]:= spheres = Graphics3D[{Sphere[{-1, -1, 0}], Sphere[{-1, 1, 0}], Sphere[{1, 1, 0}],
    Sphere[{1, -1, 0}]], Axes -> True, AxesLabel -> {"x", "y", "z"}]
```



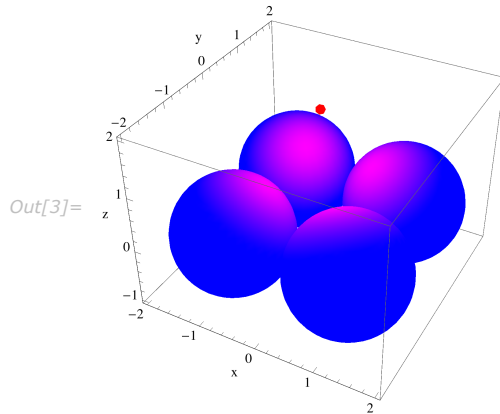
This shows the result of adding ambient light, and removing all point light sources. Note the `Lighting` option takes a list of light sources.

```
In[2]:= Show[spheres, Lighting -> {"Ambient", Blue}]
```



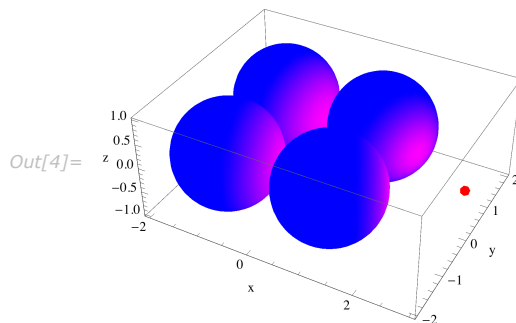
This adds a single point light source positioned at the red point. The lights are combined as appropriate.

```
In[3]:= Show[{spheres, Graphics3D[{Red, PointSize[Large], Point[{0, 0, 2}]}]},  
  Lighting -> {"Ambient", Blue}, {"Point", Red, {0, 0, 2}}]
```



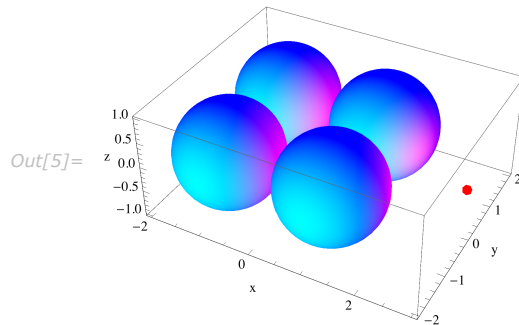
Objects do not block light sources or cast shadows, so all objects in a scene will be lit evenly by light sources.

```
In[4]:= Show[{spheres, Graphics3D[{Red, PointSize[Large], Point[{3, 0, 0}]}]},  
  Lighting -> {"Ambient", Blue}, {"Point", Red, {3, 0, 0}}]
```



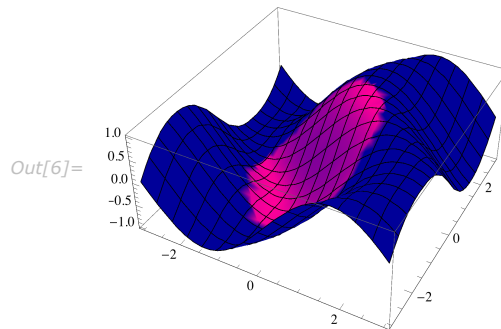
This adds a directional green light shining from the negative y direction, effectively an infinite distance away.

```
In[5]:= Show[%, Lighting -> {{ "Ambient", Blue},
                               { "Point", Red, {3, 0, 0}}, { "Directional", Green, {{0, 0, 0}, {0, 1, 0}}}]
```



This shows a spotlight positioned above the plot, combined with ambient lighting.

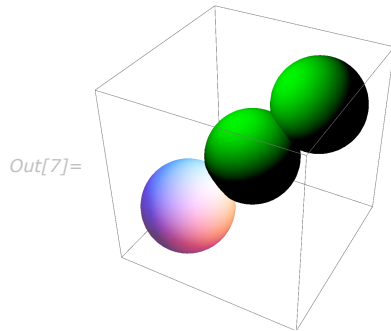
```
In[6]:= Plot3D[Sin[x + Sin[y]], {x, -3, 3}, {y, -3, 3}, Lighting ->
          {{ "Ambient", RGBColor[0, 0, .6]}, { "Spot", Red, {{0, 0, 5}, {0, 0, 0}}, 15 Degree}}]
```



The `Lighting` option controls the lighting of all objects in a scene when used as an option to `Graphics3D` or `Show`. `Lighting` can also be used inline as a directive which specifies lighting for particular objects. The `Lighting` directive replaces the inherited lighting specifications.

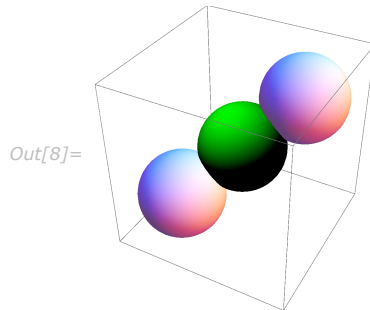
The `Lighting` directive replaces the default value of `Lighting` for the two spheres after the directive.

```
In[7]:= Graphics3D[{Sphere[{0, 0, 0}],
  Lighting -> {{\"Point\", Green, {0, 0, 5}}}, Sphere[{1, 1, 1}], Sphere[{2, 2, 2}]}
```



This example uses list braces to restrict the effect of the `Lighting` directive to the middle sphere.

```
In[8]:= Graphics3D[{Sphere[{0, 0, 0}],
  {Lighting -> {{\"Point\", Green, {0, 0, 5}}}, Sphere[{1, 1, 1}]}, Sphere[{2, 2, 2}]}
```



The perceived color of a polygon depends not only on the light which falls on the polygon, but also on how the polygon reflects that light. You can use the graphics directives `RGBColor`, `Specularity`, and `Glow` to specify the way that polygons reflect or emit light.

If you do not explicitly use these coloring directives, *Mathematica* effectively assumes that all polygons have matte white surfaces. Thus the polygons reflect light of any color incident on them, and do so equally in all directions. This is an appropriate model for materials such as uncoated white paper.

Using `RGBColor`, `Specularity`, and `Glow`, however, you can specify more complicated models. These directives separately specify three kinds of light emission: *diffuse reflection*, *specular reflection*, and *glow*.

In diffuse reflection, light incident on a surface is scattered equally in all directions. When this kind of reflection occurs, a surface has a "dull" or "matte" appearance. Diffuse reflectors obey Lambert's law of light reflection, which states that the intensity of reflected light is $\cos(\alpha)$ times the intensity of the incident light, where α is the angle between the incident light direction and the surface normal vector. Note that when $\alpha > 90^\circ$, there is no reflected light.

In specular reflection, a surface reflects light in a mirror-like way. As a result, the surface has a "shiny" or "gloss" appearance. With a perfect mirror, light incident at a particular angle is reflected at exactly the same angle. Most materials, however, scatter light to some extent, and so lead to reflected light that is distributed over a range of angles. *Mathematica* allows you to specify how broad the distribution is by giving a *specular exponent*, defined according to the Phong lighting model. With specular exponent n , the intensity of light at an angle θ away from the mirror reflection direction is assumed to vary like $\cos(\theta)^n$. As $n \rightarrow \infty$, therefore, the surface behaves like a perfect mirror. As n decreases, however, the surface becomes less "shiny", and for $n=0$, the surface is a completely diffuse reflector. Typical values of n for actual materials range from about 1 to several hundred.

Glow is light radiated from a surface at a certain color and intensity of light that is independent of incident light.

Most actual materials show a mixture of diffuse and specular reflection, and some objects glow in addition to reflecting light. For each kind of light emission, an object can have an intrinsic color. For diffuse reflection, when the incident light is white, the color of the reflected light is the material's intrinsic color. When the incident light is not white, each color component in the reflected light is a product of the corresponding component in the incident light and in the intrinsic color of the material. Similarly, an object may have an intrinsic specular reflection color, which may be different from its diffuse reflection color, and the specularly reflected light is a component-wise product of the incident light and the intrinsic specular color. For glow, the color emitted is determined by intrinsic properties alone, with no dependence on incident light.

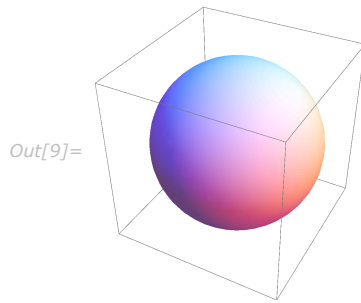
In *Mathematica*, you can specify light properties by giving any combination of diffuse reflection, specular reflection, and glow directives. To get no reflection of a particular kind, you may give the corresponding intrinsic color as `Black`, or `GrayLevel[0]`. For materials that are effectively "white", you can specify intrinsic colors of the form `GrayLevel[a]`, where a is the reflectance or albedo of the surface.

<code>GrayLevel[a]</code>	matte surface with albedo a
<code>RGBColor[r,g,b]</code>	matte surface with intrinsic color
<code>Specularity[spec,n]</code>	surface with specularity $spec$ and specular exponent n ; $spec$ can be a number between 0 and 1 or an <code>RGBColor</code> specification
<code>Glow[col]</code>	glowing surface with color col

Specifying surface properties of lighted objects.

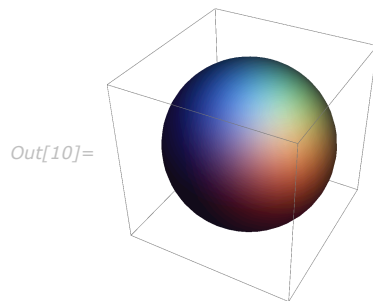
This shows a sphere with the default matte white surface, illuminated by several colored light sources.

```
In[9]:= Graphics3D[Sphere[]]
```



This makes the sphere have low diffuse reflectance, but high specular reflectance. As a result, the sphere has a "specular highlight" near the light sources, and is quite dark elsewhere.

```
In[10]:= Graphics3D[{GrayLevel[0.2], Specularity[0.8, 5], Sphere[]}]
```



When you set up light sources and surface colors, it is important to make sure that the total intensity of light reflected from a particular polygon is never larger than 1. You will get strange effects if the intensity is larger than 1.

Labeling Three-Dimensional Graphics

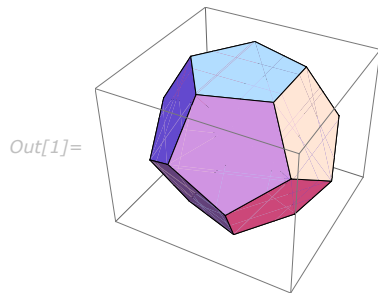
Mathematica provides various options for labeling three-dimensional graphics. Some of these options are directly analogous to those for two-dimensional graphics, discussed in "Labeling Two-Dimensional Graphics". Others are different.

<code>Boxed->True</code>	draw a cuboidal bounding box around the graphics (default)
<code>Axes->True</code>	draw x , y and z axes on the edges of the box
<code>Axes->{False,False,True}</code>	draw the z axis only
<code>FaceGrids->All</code>	draw grid lines on the faces of the box
<code>PlotLabel->text</code>	give an overall label for the plot

Some options for labeling three-dimensional graphics.

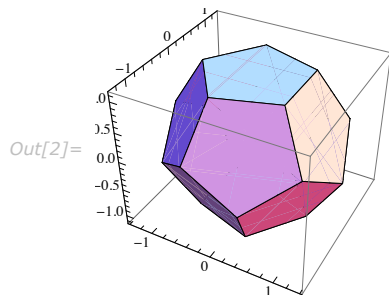
The default for `Graphics3D` is to include a box, but no other forms of labeling.

```
In[1]:= Graphics3D[PolyhedronData["Dodecahedron", "Faces"]]
```



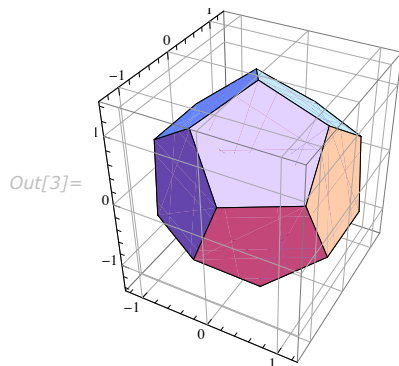
Setting `Axes -> True` adds x , y and z axes.

```
In[2]:= Show[%, Axes -> True]
```



This adds grid lines to each face of the box.

```
In[3]:= Show[%, FaceGrids -> All]
```



`BoxStyle->style`

specify the style for the box

`AxesStyle->style`

specify the style for axes

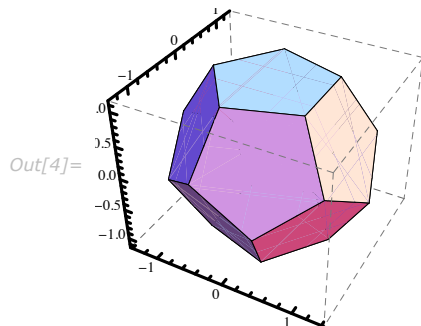
`AxesStyle->{xstyle, ystyle, zstyle}`

specify separate styles for each axis

Style options.

This makes the box dashed, and draws axes which are thicker than normal.

```
In[4]:= Graphics3D[PolyhedronData["Dodecahedron", "Faces"],
  BoxStyle -> Dashing[{0.02, 0.02}], Axes -> True, AxesStyle -> Thickness[0.01]]
```



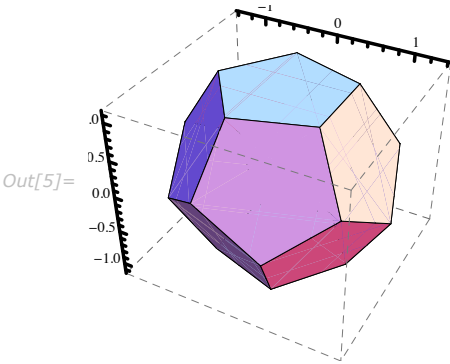
By setting the option `Axes -> True`, you tell *Mathematica* to draw axes on the edges of the three-dimensional box. However, for each axis, there are in principle four possible edges on which it can be drawn. The option `AxesEdge` allows you to specify on which edge to draw each of the axes.

<code>AxesEdge->Automatic</code>	use an internal algorithm to choose where to draw all axes
<code>AxesEdge->{<i>xspec</i>,<i>yspec</i>,<i>zspec</i>}</code>	give separate specifications for each of the <i>x</i> , <i>y</i> and <i>z</i> axes
<code>None</code>	do not draw this axis
<code>Automatic</code>	decide automatically where to draw this axis
<code>{<i>dir_i</i>,<i>dir_j</i>}</code>	specify on which of the four possible edges to draw this axis

Specifying where to draw three-dimensional axes.

This draws the *x* on the edge with larger *y* and *z* coordinates, draws no *y* axis, and chooses automatically where to draw the *z* axis.

```
In[5]:= Show[%, Axes -> True, AxesEdge -> {{1, 1}, None, Automatic}]
```



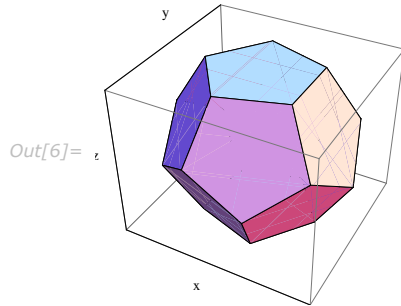
When you draw the *x* axis on a three-dimensional box, there are four possible edges on which the axis can be drawn. These edges are distinguished by having larger or smaller *y* and *z* coordinates. When you use the specification `{diry,dirz}` for where to draw the *x* axis, you can set the *dir_i* to be +1 or -1 to represent larger or smaller values for the *y* and *z* coordinates.

<code>AxesLabel->None</code>	give no axis labels
<code>AxesLabel-><i>zlabel</i></code>	put a label on the <i>z</i> axis
<code>AxesLabel->{<i>xlabel</i>,<i>ylabel</i>,<i>zlabel</i>}</code>	put labels on all three axes

Axis labels in three-dimensional graphics.

You can use `AxesLabel` to label edges of the box, without necessarily drawing scales on them.

```
In[6]:= Show[PolyhedronData["Dodecahedron", "Image"],
  Axes -> True, AxesLabel -> {"x", "y", "z"}, Ticks -> None]
```



<code>Ticks -> None</code>	draw no tick marks
<code>Ticks -> Automatic</code>	place tick marks automatically
<code>Ticks -> {xticks, yticks, zticks}</code>	tick mark specifications for each axis

Settings for the `Ticks` option.

You can give the same kind of tick mark specifications in three dimensions as were described for two-dimensional graphics in "Labeling Two-Dimensional Graphics".

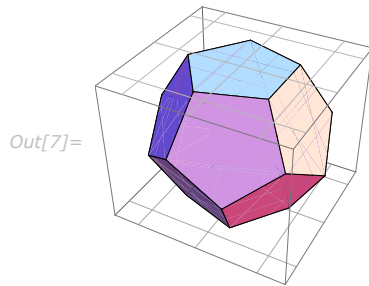
<code>FaceGrids -> None</code>	draw no grid lines on faces
<code>FaceGrids -> All</code>	draw grid lines on all faces
<code>FaceGrids -> {face₁, face₂, ...}</code>	draw grid lines on the faces specified by the <i>face_i</i>
<code>FaceGrids -> {{face₁, {xgrid₁, ygrid₁}}, ...}</code>	use <i>xgrid_i</i> , <i>ygrid_i</i> to determine where and how to draw grid lines on each face

Drawing grid lines in three dimensions.

Mathematica allows you to draw grid lines on the faces of the box that surrounds a three-dimensional object. If you set `FaceGrids -> All`, grid lines are drawn in gray on every face. By setting `FaceGrids -> {face1, face2, ...}` you can tell *Mathematica* to draw grid lines only on specific faces. Each face is specified by a list $\{dir_x, dir_y, dir_z\}$, where two of the dir_i must be 0, and the third one is +1 or -1. For each face, you can also explicitly tell *Mathematica* where and how to draw the grid lines, using the same kind of specifications as you give for the `GridLines` option in two-dimensional graphics.

This draws grid lines only on the top and bottom faces of the box.

```
In[7]:= Show[PolyhedronData["Dodecahedron", "Image"], FaceGrids -> {{0, 0, 1}, {0, 0, -1}}]
```



Efficient Representation of Many Primitives

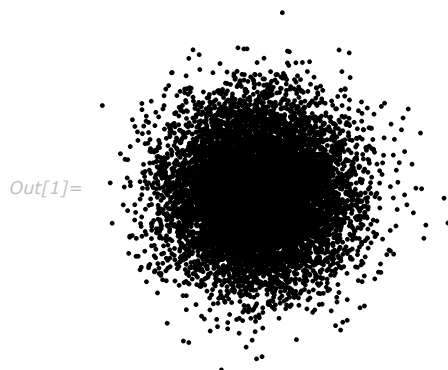
<code>Point[{pt_1, pt_2, ...}]</code>	a multipoint consisting of points at pt_1 , pt_2 , ...
<code>Line[{$line_1$, $line_2$, ...}]</code>	a multiline consisting of lines $line_1$, $line_2$, ...
<code>Polygon[{$poly_1$, $poly_2$, ...}]</code>	a multipolygon consisting of polygons $poly_1$, $poly_2$, ...

Primitives which can take multiple elements.

Some primitives have multi-element forms that can be processed and rendered more quickly by the *Mathematica* front end than the equivalent individual primitives. For large numbers of primitives, using the multi-element forms can also significantly reduce the sizes of notebook files. Notebooks that use multi-element forms can be less than half the size of those that do not, and render up to ten times faster.

Here is a multipoint random distribution.

```
In[1]:= Graphics[Point[Table[RandomReal[NormalDistribution[0, 1], 2], {10 000}]]]
```



GraphicsComplex[
 $\{pt_1, pt_2, \dots\}, data]$

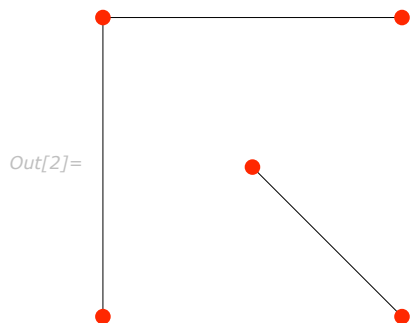
a graphics complex in which coordinates given as integers i in graphics primitives in *data* are taken to be pt_i

Primitive for sharing coordinate data among primitives.

When many primitives share the same coordinate data, as in meshes and graphs, further efficiency can be gained by using **GraphicsComplex** to factor out the coordinate data. The output of *Mathematica's* surface- and graph-plotting functions typically use this representation.

Here is a structure of points and lines that share coordinates.

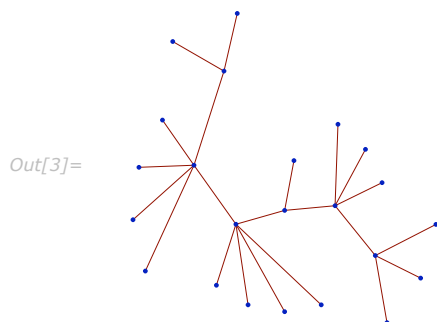
**In[2]:= Graphics[GraphicsComplex[{{0, 0}, {0, 1}, {1, 1}, {1, 0}, {.5, .5}},
 {Line[{{1, 2, 3}, {5, 4}}], Red, PointSize[.05], Point[{1, 2, 3, 4, 5}]}]]**



In addition to being efficient, **GraphicsComplex** is useful interactively. Primitives that share coordinates stay connected when one of them is dragged.

Because the output of **GraphPlot** is a **GraphicsComplex**, the graph stays connected when any part of it is dragged.

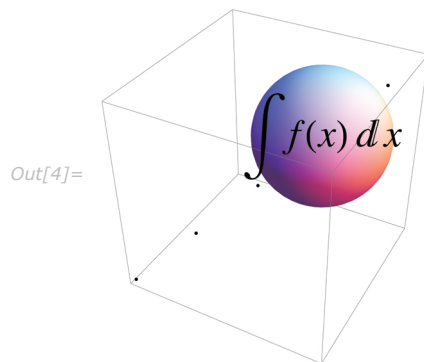
In[3]:= GraphPlot[Table[i → Mod[i² - 199, 10], {i, 1, 22}]]



Any primitive may be used within a `GraphicsComplex`, and `GraphicsComplex` can be used in both 2D and 3D graphics. Within `GraphicsComplex`, coordinate positions in primitives are replaced by indices into the coordinate data in the `GraphicsComplex`.

This `GraphicsComplex` combines several types of primitives.

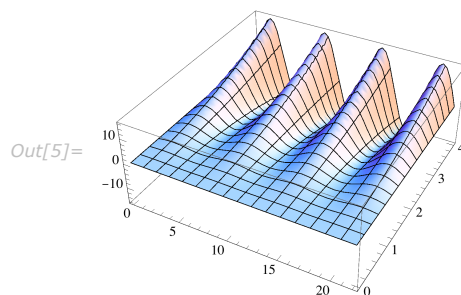
```
In[4]:= Graphics3D[GraphicsComplex[Table[{i, i, i}, {i, 1, 5}], {
  Sphere[4, 1.5],
  Text[Style[ $\int f(x) dx$ , 36], 4],
  Point[Range[5]]
}]]
```



`GraphicsComplex` is especially useful for representing meshes of polygons. By using `GraphicsComplex`, numerical errors that could cause gaps between adjacent polygons are avoided.

The output of `Plot3D` is a `GraphicsComplex`.

```
In[5]:= Plot3D[Sin[x] y^2, {x, 0, 22}, {y, 0, 4}]
```



Formats for Text in Graphics

BaseStyle→*value*

an option for the text style in a graphic

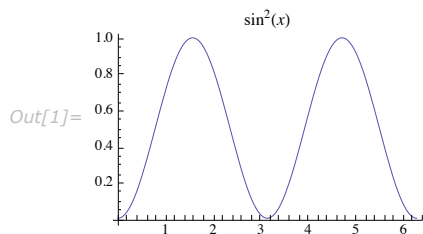
FormatType→*value*

an option for the text format type in a graphic

Specifying formats for text in graphics.

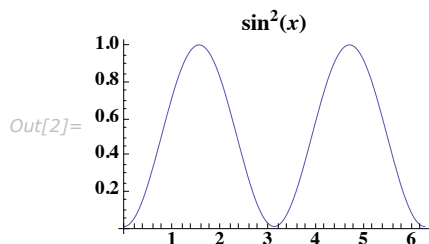
Here is a plot with default settings for all formats.

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



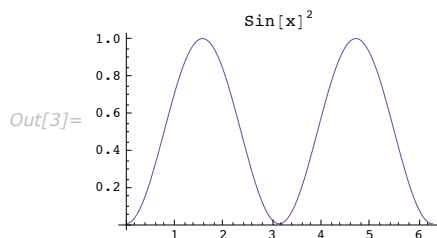
Here is the same plot, but now using a 12-point bold font.

```
In[2]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Sin[x]^2,
  BaseStyle -> {FontWeight -> "Bold", FontSize -> 12}]
```



This uses StandardForm rather than TraditionalForm .

```
In[3]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Sin[x]^2, FormatType -> StandardForm]
```

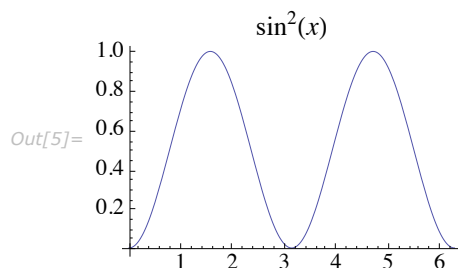


This tells *Mathematica* what default text style to use for all subsequent plots.

```
In[4]:= SetOptions[Plot, BaseStyle -> {FontFamily -> "Times", FontSize -> 14}];
```

Now all the text is in 14-point Times font.

```
In[5]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Sin[x]^2]
```



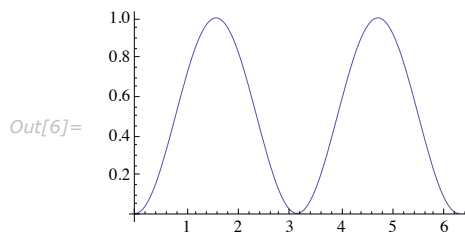
"style"	a named style in your current stylesheet
FontSize->n	the size of font to use in printer's points
FontSlant->"Italic"	use an italic font
FontWeight->"Bold"	use a bold font
FontFamily->"name"	specify the name of the font family to use (e.g. "Times", "Courier", "Helvetica")

Typical elements used in the setting for `BaseStyle`.

If you use the standard notebook front end for *Mathematica*, then you can set `BaseStyle` to be the name of a style defined in your current notebook's stylesheet. You can also explicitly specify how text should be formatted by using options such as `FontSize` and `FontFamily`. Note that `FontSize` gives the absolute size of the font to use, measured in units of printer's points, with one point being $\frac{1}{72}$ inches. If you resize a plot whose font size is specified as a number, the text in it will not by default change size: to get text of a different size you must explicitly specify a new value for the `FontSize` option. If you resize a plot whose font size is specified as a scaled quantity, the font will scale as the plot is resized. With `FontSize -> Scaled[s]`, the effective font size will be s scaled units in the plot.

Now all the text resizes as the plot is resized.

```
In[6]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, BaseStyle -> {FontSize -> Scaled[.05]}]
```

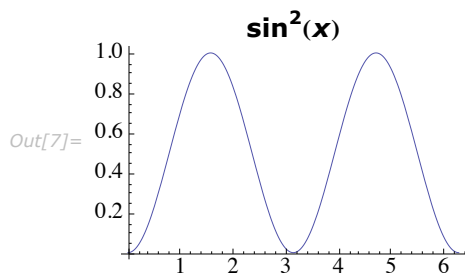


<code>Style[expr, "style"]</code>	output <i>expr</i> in the specified style
<code>Style[expr, options]</code>	output <i>expr</i> using the specified font and style options
<code>StandardForm[expr]</code>	output <i>expr</i> in StandardForm

Changing the formats of individual pieces of output.

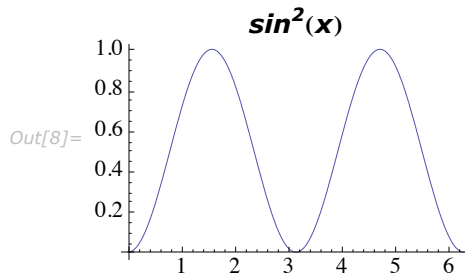
This outputs the plot label using the section heading style in your current notebook.

```
In[7]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Style[Sin[x]^2, "Section"]]
```



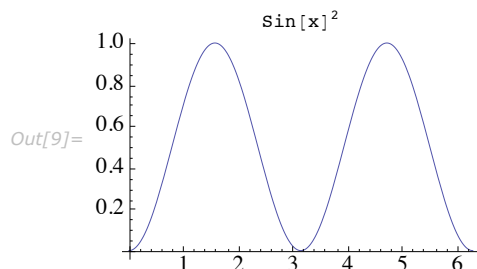
This uses the section heading style, but modified to be in italics.

```
In[8]:= Plot[Sin[x]^2, {x, 0, 2 Pi}, PlotLabel -> Style[Sin[x]^2, "Section", FontSlant -> "Italic"]]
```



This produces `StandardForm` output, with a 12-point font.

```
In[9]:= Plot[Sin[x]^2, {x, 0, 2 Pi},
  PlotLabel -> Style[StandardForm[Sin[x]^2], FontSize -> 12]]
```



You should realize that the ability to refer to styles such as "Section" depends on using the standard *Mathematica* notebook front end. Even if you are just using a text-based interface to *Mathematica*, however, you can still specify formatting of text in graphics using options such as `FontSize`. The complete collection of options that you can use is given in "Text and Font Options".

Graphics Primitives for Text

With the `Text` graphics primitive, you can insert text at any position in two- or three-dimensional *Mathematica* graphics. Unless you explicitly specify a style or font using `style`, the text will be given in the graphic's base style.

<code>Text[expr, {x, y}]</code>	text centered at the point $\{x, y\}$
<code>Text[expr, {x, y}, {-1, 0}]</code>	text with its left-hand end at $\{x, y\}$
<code>Text[expr, {x, y}, {1, 0}]</code>	right-hand end at $\{x, y\}$
<code>Text[expr, {x, y}, {0, -1}]</code>	centered above $\{x, y\}$
<code>Text[expr, {x, y}, {0, 1}]</code>	centered below $\{x, y\}$
<code>Text[expr, {x, y}, {dx, dy}]</code>	text positioned so that $\{x, y\}$ is at relative coordinates $\{dx, dy\}$ within the box that bounds the text
<code>Text[expr, {x, y}, {dx, dy}, {0, 1}]</code>	text oriented vertically to read from bottom to top
<code>Text[expr, {x, y}, {dx, dy}, {0, -1}]</code>	text that reads from top to bottom
<code>Text[expr, {x, y}, {dx, dy}, {-1, 0}]</code>	text that is upside-down

Two-dimensional text.

This generates five pieces of text, and displays them in a plot.

```
In[1]:= Show[Graphics[Table[Text[Expand[(1 + x)^n], {n, n}], {n, 5}]], PlotRange -> All]
```

$$x^5 + 5x^4 + 10x^3 + 10x^2 + 5x + 1$$

$$x^4 + 4x^3 + 6x^2 + 4x + 1$$

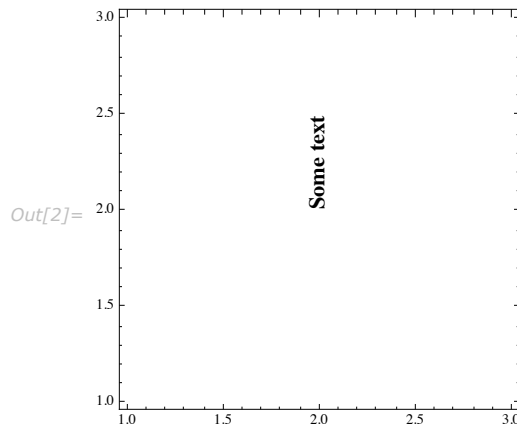
$$\text{Out[1]= } x^3 + 3x^2 + 3x + 1$$

$$x^2 + 2x + 1$$

$$x + 1$$

Here is some vertically oriented text with its left-hand side at the point {2, 2}.

```
In[2]:= Show[Graphics[Text[StyleForm["Some text", FontSize -> 14, FontWeight -> "Bold"],
{2, 2}, {-1, 0}, {0, 1}], Frame -> True]
```



When you specify an offset for text, the relative coordinates that are used are taken to run from -1 to 1 in each direction across the box that bounds the text. The point {0, 0} in this coordinate system is defined to be center of the text. Note that the offsets you specify need not lie in the range -1 to 1.

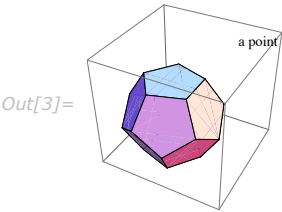
Note that you can specify the color of a piece of text by preceding the `Text` graphics primitive with an appropriate `RGBColor` or other graphics directive.

<code>Text[expr, {x, y, z}]</code>	text centered at the point {x, y, z}
<code>Text[expr, {x, y, z}, {sdx, sdy}]</code>	text with a two-dimensional offset

Three-dimensional text.

This puts text at the specified position in three dimensions.

```
In[3]:= Graphics3D[
  {PolyhedronData["Dodecahedron", "Faces"], Text["a point", {2, 2, 2}, {1, 1}]}]
```



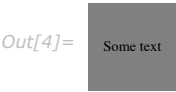
Note that when you use text in three-dimensional graphics, *Mathematica* assumes that the text is never hidden by any polygons or other objects.

option name	default value	
Background	None	background color
BaseStyle	{ }	style or font specification
FormatType	StandardForm	format type

Options for `Text`.

By default the text is just put straight on top of whatever graphics have already been drawn.

```
In[4]:= Graphics[
  {{GrayLevel[0.5], Rectangle[{0, 0}, {1, 1}]}, Text["Some text", {0.5, 0.5}]}]
```



Now there is a rectangle with the background color of the whole plot enclosing the text.

```
In[5]:= Graphics[{{GrayLevel[0.5], Rectangle[{0, 0}, {1, 1}]},
  Text["Some text", {0.5, 0.5}, Background -> Automatic]}]
```



The Representation of Sound

"Sound" describes how you can take functions and lists of data and produce sounds from them. Here we discuss how sounds are represented in *Mathematica*.

Mathematica treats sounds much like graphics. In fact, *Mathematica* allows you to combine graphics with sound to create pictures with "sound tracks".

In analogy with graphics, sounds in *Mathematica* are represented by symbolic sound objects. The sound objects have head `Sound`, and contain a list of sound primitives, which represent sounds to be played in sequence.

`Sound [ {  $s_1, s_2, \dots$  } ]`

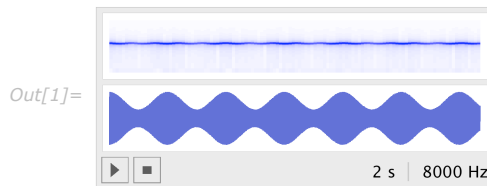
a sound object containing a list of sound primitives

The structure of a sound object.

The functions `Play` and `ListPlay` discussed in "Sound" return sound objects.

`Play` returns a `Sound` object. On appropriate computer systems, it also produces sound.

```
In[1]:= Play[(2 + Cos[20 t]) * Sin[3000 t + 2 Sin[50 t]], {t, 0, 2}]
```



The `Sound` object contains a `SampledSoundFunction` primitive which uses a compiled function to generate amplitude samples for the sound.

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

```
Out[2]//Short= Sound[SampledSoundFunction[CompiledFunction[[_Integer],
{<<2>>}, <<3>>, Evaluate], 16384, 8192]]
```

`SampledSoundList [ {  $a_1, a_2, \dots$  },  $r$  ]`

a sound with a sequence of amplitude levels, sampled at rate r

`SampledSoundFunction [  $f, n, r$  ]`

a sound whose amplitude levels sampled at rate r are found by applying the function f to n successive integers

`SoundNote [  $n, t, \text{"style"}$  ]`

a note-like sound with note n , time specification t , with the specified style

Mathematica sound primitives.

At the lowest level, all sounds in *Mathematica* are represented as a sequence of amplitude samples, or as a sequence of MIDI events. In `SampledSoundList`, these amplitude samples are given explicitly in a list. In `SampledSoundFunction`, however, they are generated when the sound is output, by applying the specified function to a sequence of integer arguments. In both cases, all amplitude values obtained must be between -1 and 1 . In `SoundNote`, a note-like sound is represented as a sequence of MIDI events that represent the frequency, duration, amplitude and styling of the note.

`ListPlay` generates `SampledSoundList` primitives, while `Play` generates `SampledSoundFunction` primitives. With the default option setting `Compiled -> True`, `Play` will produce a `SampledSoundFunction` object containing a `CompiledFunction`.

Once you have generated a sound object containing various sound primitives, you must then output it as a sound. Much as with graphics, the basic scheme is to take the *Mathematica* representation of the sound, and convert it to a lower-level form that can be handled by an external program, such as a *Mathematica* front end.

The low-level representation of sampled sound used by *Mathematica* consists of a sequence of hexadecimal numbers specifying amplitude levels. Within *Mathematica*, amplitude levels are given as approximate real numbers between -1 and 1 . In producing the low-level form, the amplitude levels are “quantized”. You can use the option `SampleDepth` to specify how many bits should be used for each sample. The default is `SampleDepth -> 8`, which yields 256 possible amplitude levels, sufficient for most purposes. The low-level representation of note-based sound is as a time-quantized byte stream of MIDI events, which specify various parameters about the note objects. The quantization of time is determined automatically at playback.

You can use the option `SampleDepth` in `Play` and `ListPlay`. In sound primitives, you can specify the sample depth by replacing the sample rate argument by the list `{rate, depth}`.

Exporting Graphics and Sounds

Mathematica allows you to export graphics and sounds in a wide variety of formats. If you use the notebook front end for *Mathematica*, then you can typically just copy and paste graphics and sounds directly into other programs using the standard mechanism available on your computer system.

<code>Export["name.ext", graphics]</code>	export graphics to a file in a format deduced from the file name
<code>Export["file", graphics, "format"]</code>	export graphics in the specified format
<code>Export["!command", graphics, "format"]</code>	export graphics to an external command
<code>Export["file", {g₁, g₂, ...}, ...]</code>	export a sequence of graphics for an animation
<code>ExportString[graphics, "format"]</code>	generate a string representation of exported graphics

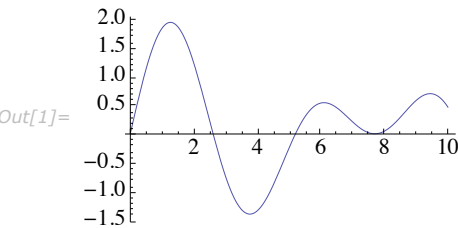
Exporting *Mathematica* graphics and sounds.

"EPS"	Encapsulated PostScript (.eps)
"PDF"	Adobe Acrobat portable document format (.pdf)
"SVG"	Scalable Vector Graphics (.svg)
"PICT"	Macintosh PICT
"WMF"	Windows metafile format (.wmf)
"TIFF"	TIFF (.tif, .tiff)
"GIF"	GIF and animated GIF (.gif)
"JPEG"	JPEG (.jpg, .jpeg)
"PNG"	PNG format (.png)
"BMP"	Microsoft bitmap format (.bmp)
"PCX"	PCX format (.pcx)
"XBM"	X window system bitmap (.xbm)
"PBM"	portable bitmap format (.pbm)
"PPM"	portable pixmap format (.ppm)
"PGM"	portable graymap format (.pgm)
"PNM"	portable anymap format (.pnm)
"DICOM"	DICOM medical imaging format (.dcm, .dic)
"AVI"	Audio Video Interleave format (.avi)

Typical graphics formats supported by *Mathematica*. Formats in the first group are resolution independent.

This generates a plot.

```
In[1]:= Plot[Sin[x] + Sin[Sqrt[2] x], {x, 0, 10}]
```



This exports the plot to a file in Encapsulated PostScript format.

```
In[2]:= Export["sinplot.eps", %]
```

```
Out[2]= sinplot.eps
```

When you export a graphic outside of *Mathematica*, you usually have to specify the absolute size at which the graphic should be rendered. You can do this using the `ImageSize` option to `Export`.

`ImageSize -> x` makes the width of the graphic be x printer's points; `ImageSize -> 72 xi` thus makes the width xi inches. The default is to produce an image that is four inches wide. `ImageSize -> {x, y}` scales the graphic so that it fits in an $x \times y$ region.

<code>ImageSize</code>	<code>Automatic</code>	absolute image size in printer's points
<code>"ImageTopOrientation"</code>	<code>Top</code>	how the image is oriented in the file
<code>ImageResolution</code>	<code>Automatic</code>	resolution in dpi for the image

Options for `Export`.

Within *Mathematica*, graphics are manipulated in a way that is completely independent of the resolution of the computer screen or other output device on which the graphics will eventually be rendered.

Many programs and devices accept graphics in resolution-independent formats such as Encapsulated PostScript (EPS). But some require that the graphics be converted to rasters or bitmaps with a specific resolution. The `ImageResolution` option for `Export` allows you to determine what resolution in dots per inch (dpi) should be used. The lower you set this resolution, the lower the quality of the image you will get, but also the less memory the image will take to store. For screen display, typical resolutions are 72 dpi and above; for printers, 300 dpi and above.

<code>"DXF"</code>	AutoCAD drawing interchange format (<code>.dxf</code>)
<code>"STL"</code>	STL stereolithography format (<code>.stl</code>)

Typical 3D geometry formats supported by *Mathematica*.

<code>"WAV"</code>	Microsoft wave format (<code>.wav</code>)
<code>"AU"</code>	μ law encoding (<code>.au</code>)
<code>"SND"</code>	sound file format (<code>.snd</code>)
<code>"AIFF"</code>	AIFF format (<code>.aif</code> , <code>.aiff</code>)

Typical sound formats supported by *Mathematica*.

Importing Graphics and Sounds

Mathematica allows you not only to export graphics and sounds, but also to import them. With `Import` you can read graphics and sounds in a wide variety of formats, and bring them into *Mathematica* as *Mathematica* expressions.

<code>Import["name.ext"]</code>	import graphics from the file <i>name.ext</i> in a format deduced from the file name
<code>Import["file", "format"]</code>	import graphics in the specified format
<code>ImportString["string", "format"]</code>	import graphics from a string

Importing graphics and sounds.

This imports an image stored in JPEG format.

```
In[1]:= g = Import["ExampleData/ocelot.jpg"]
```

Out[1]=



This shows an array of four copies of the image.

```
In[2]:= GraphicsGrid[{{g, g}, {g, g}}]
```

Out[2]=



`Import` yields expressions with different structures depending on the type of data it reads. Typically you will need to know the structure if you want to manipulate the data that is returned.

<code>Graphics [primitives, opts]</code>	resolution-independent graphics
<code>Graphics [Raster[data], opts]</code>	resolution-dependent bitmap images
<code>{graphics₁, graphics₂, ...}</code>	animated graphics
<code>Sound [SampledSoundList[data, r]]</code>	sounds

Structures of expressions returned by `Import`.

This shows the overall structure of the graphics object imported above.

```
In[3]:= Shallow[InputForm[g]]
```

```
Out[3]//Shallow= Graphics[Raster[<< 4 >>], Rule[<< 2 >>], Rule[<< 2 >>]]
```

This extracts the array of pixel values used.

```
In[4]:= d = g[[1, 1]];
```

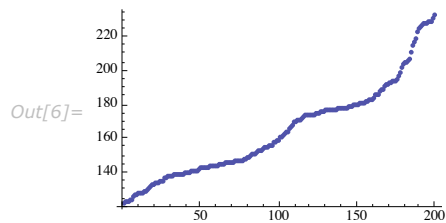
Here are the dimensions of the array.

```
In[5]:= Dimensions[d]
```

```
Out[5]= {200, 200}
```

This shows the distribution of pixel values.

```
In[6]:= ListPlot[Sort[Flatten[d]]]
```



This shows a transformed version of the image.

```
In[7]:= Graphics[Raster[d^2 / Max[d^2]]]
```

Out[7]=



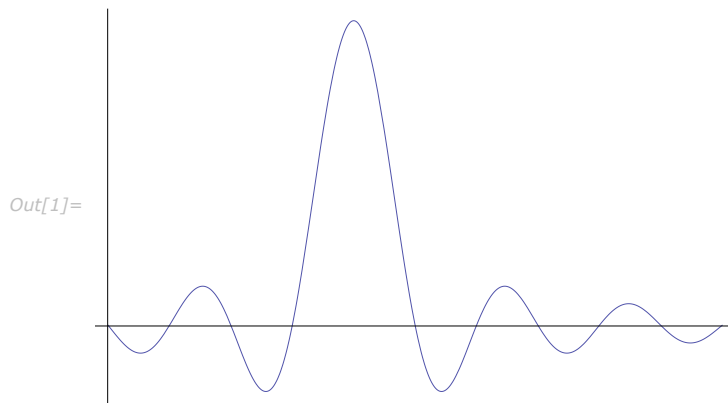
Editing *Mathematica* Graphics

Introduction to Editing *Mathematica* Graphics

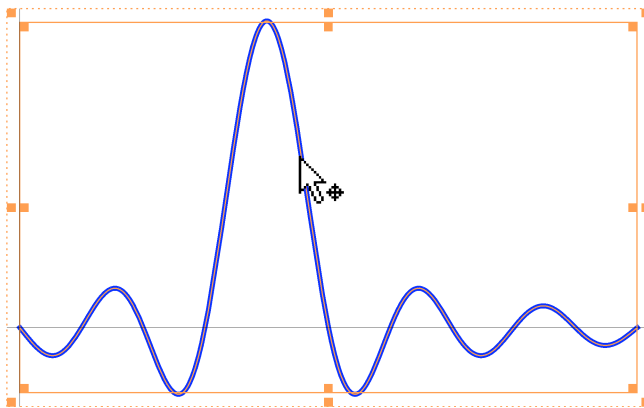
An Example of Editing Graphics

The following graph represents an impulse response of an ideal Low Pass Filter (LPF). This graph illustrates some of the ways of interacting with graphics. Details on each topic follow in the other parts of "Interactive Graphics".

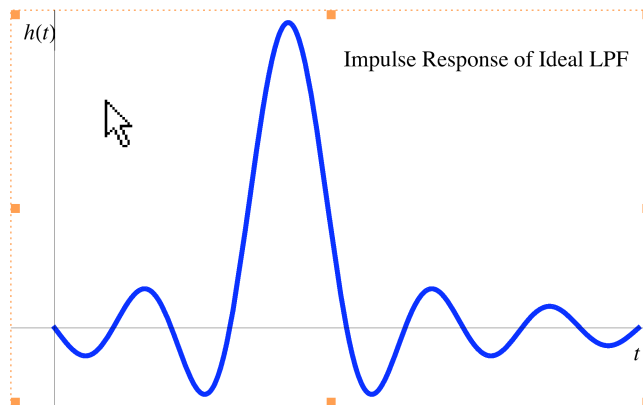
`In[1]:= Plot[Sin[Pi (t - 4)] / (Pi (t - 4)), {t, 0, 10}, PlotRange -> All, Ticks -> None]`



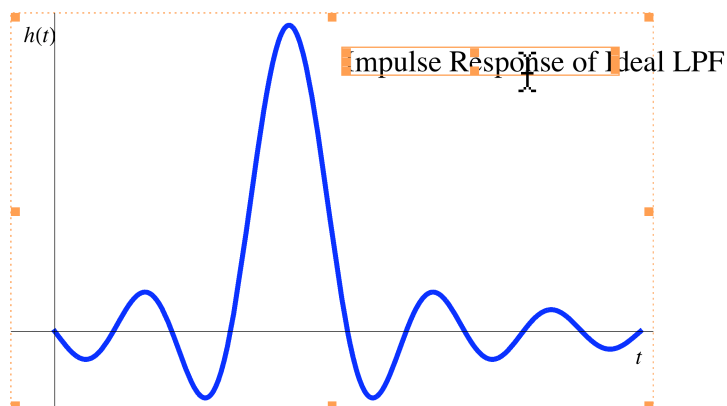
Select the curve and change its color and thickness using the **Graphics Inspector**.



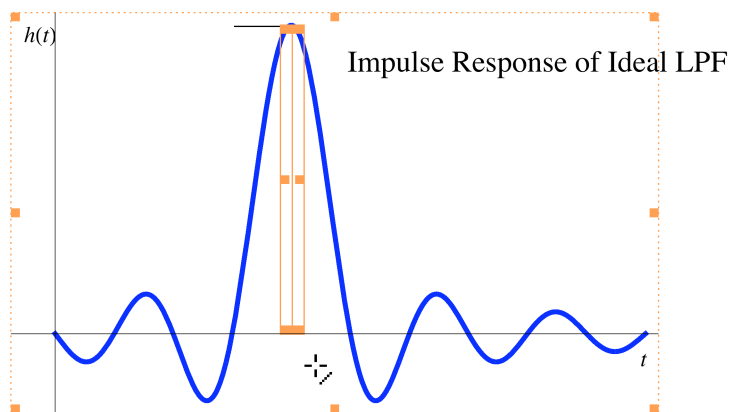
Using the **Text** tool and the **TraditionalForm Text** tool, add a plot label and axis labels.



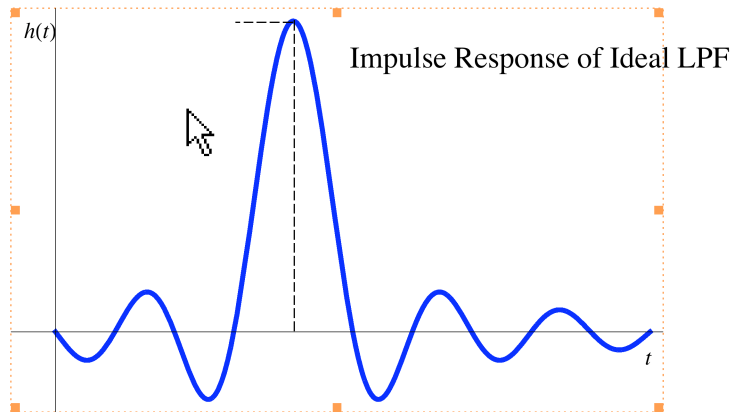
You can edit the title to change its font, color, size, and face.



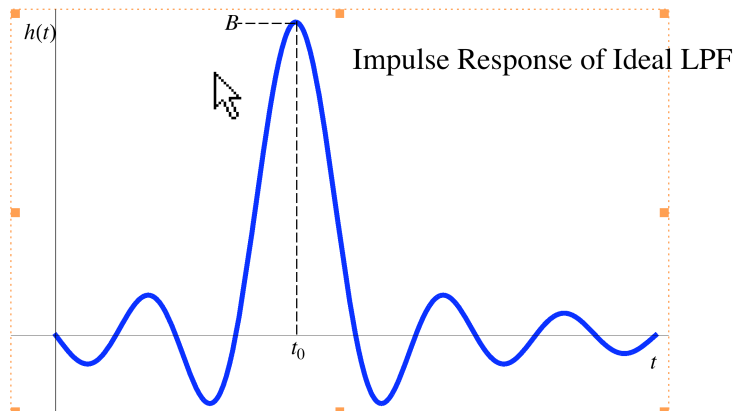
Draw vertical and horizontal lines to the maximum point with the **Line** tool.



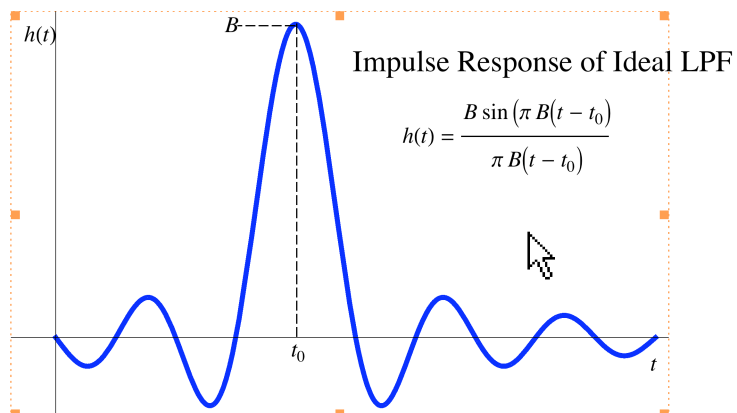
Select the lines and make them dashed using the **Graphics Inspector**.



Label the point with the **TraditionalForm Text** tool.



Add the formula for the curve.

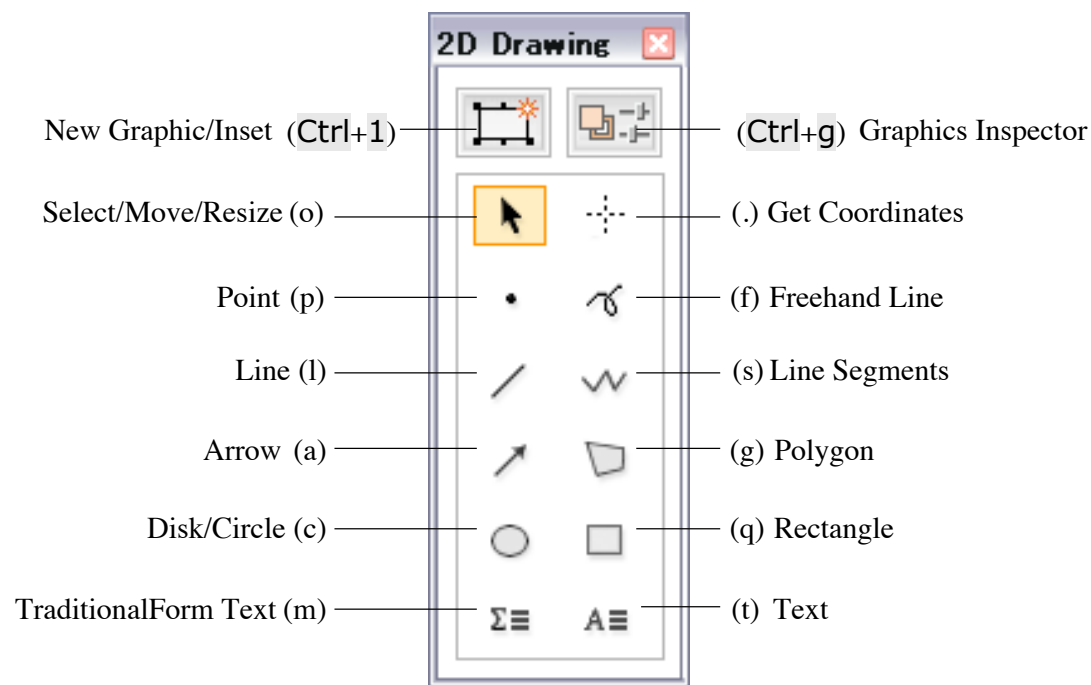


Drawing Tools

To Open the Graphics Palette:

Type **Ctrl+T** or choose **Graphics ► Drawing Tools**.

For more information on each tool, click the words pointing into the palette.



To Select a Tool:

Do one of the following:

- Click a tool icon on the palette.
- With a graphic selected, type one of these letters:
o, p, f, l, s, a, g, c, q, m, t

Persistence of Tools

Do one of the following:

- Click a tool button to use a tool once.
After the single use, the tool will automatically revert to the **Selection** tool.
- Double-click a tool button to keep using the tool.

Tools

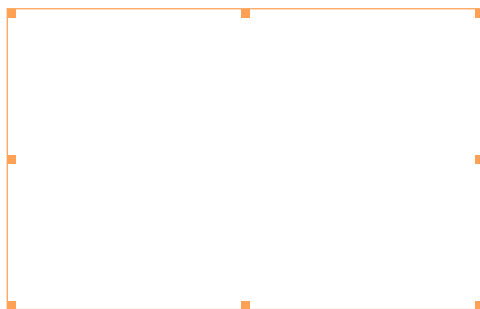
New Graphic Tool



Do one of the following:

- Type **Command +1**.
- Click the **New Graphic** button on the palette.
- Choose **Graphics ► New Graphic**.

A blank drawing area with a bounding box appears.



Selection Tool

You can use the **Selection** tool to select a graphics primitive as a whole.

The selection is indicated by a frame with handles.

You can scale the selection by dragging a handle.

To scale equally in both directions, **Shift**+drag a handle.

You can move an object by dragging it.

You can remove or add an object to a selection with **Shift**+click.

Drag out a rectangle to select all the objects within it.

When an object is completely underneath another one, dragging makes it visible.

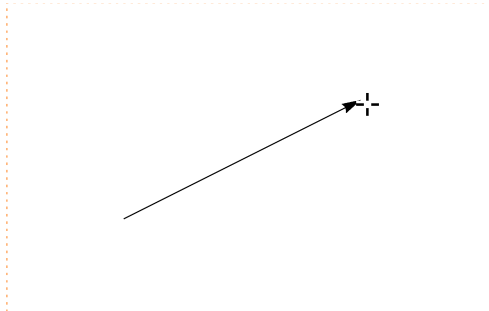
You can then drag it by dragging its highlight.

For more information on how to use the **Selection** tool, see "Interactive Graphics: Selecting".

Draw Arrow Tool

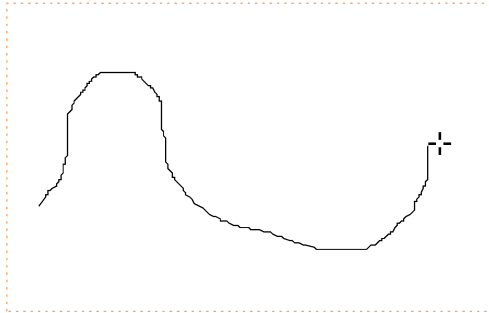
Click the **Draw Arrow** tool and drag the pointer () to draw an arrow.

Hold down the **Shift** key to draw the arrow horizontally or vertically.



Draw Freehand Tool

Click the **Draw Freehand** tool button and drag the pointer () to draw a curve.



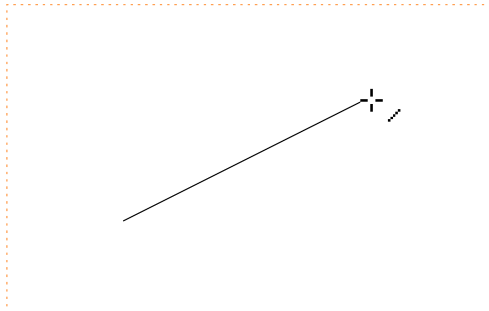
Double-click the **Draw Freehand** tool to draw multiple curves.



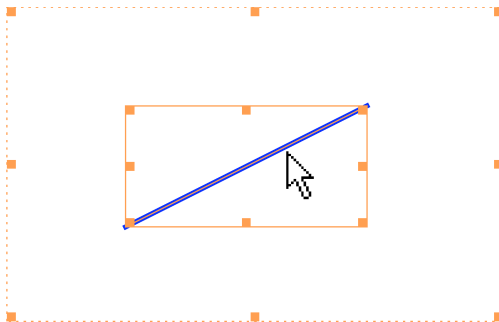
Draw Line Tool

Click the **Draw Line** tool and drag the pointer () to draw a single line.

Hold down the Shift key to draw a horizontal or vertical line.



You can set line styles with the **Graphics Inspector** palette.

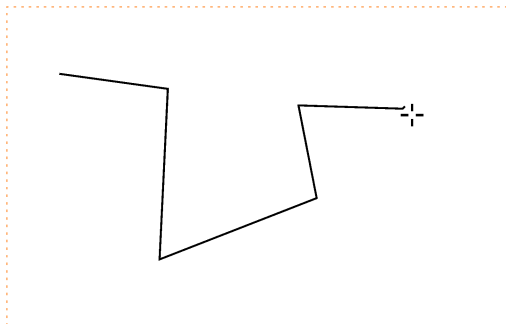


Draw Line Segments Tool

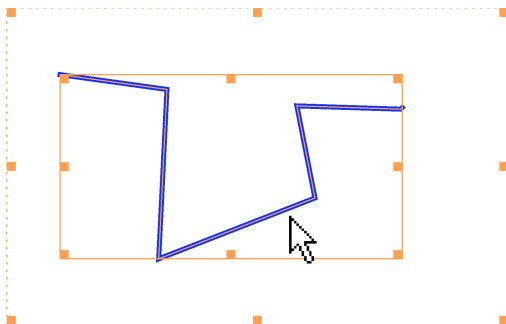
Click the **Draw Line Segments** tool and drag the pointer () to draw a multi-segment line.

You get a new segment after each click.

To stop, double-click the last point or single-click the first point.

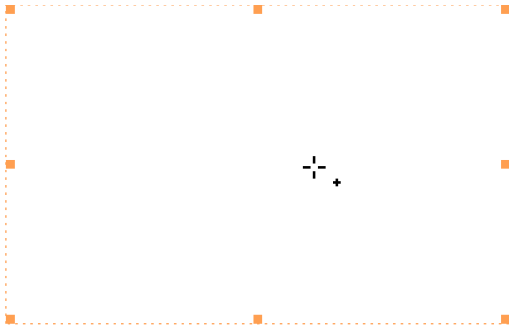


You can set the line style with the **Graphics Inspector** palette.

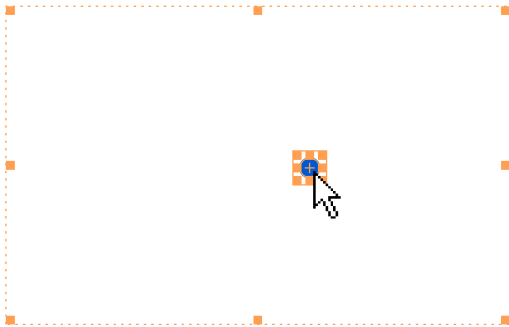


Draw Point Tool

Click the **Draw Point** tool and then click to draw a point.

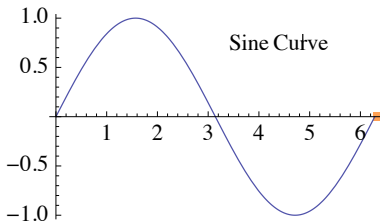
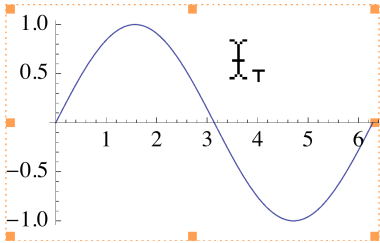


You can set point styles with the **Graphics Inspector**.



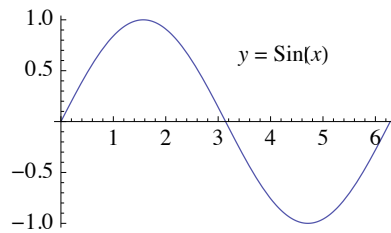
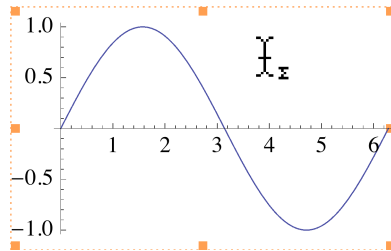
Place Text Tool

You can place text in a graphic with the **Place Text** tool.



Place TraditionalForm Text Tool

You can place TraditionalForm text in a graphic with the **Place TraditionalForm Text** tool.



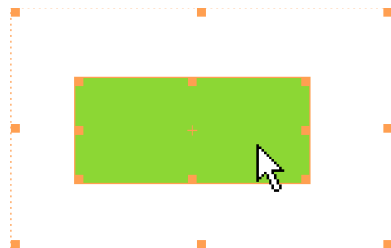
Draw Rectangle Tool

Click the **Draw Rectangle** tool and drag the pointer () to draw a rectangle.

Hold down the Shift key to draw squares.



You can set face and edge styles with the **Graphics Inspector** palette.



Draw Polygon Tool

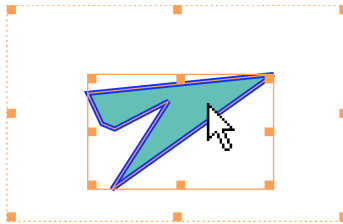
Click the **Draw Polygon** tool and drag the pointer () to draw a polygon.

You get a new segment after each click.

To stop, double-click the last point or single-click the first point.

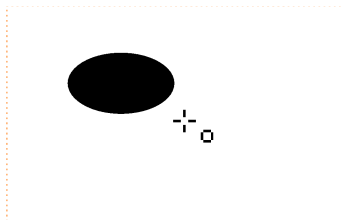


You can set face and edge styles with the **Graphics Inspector** palette.

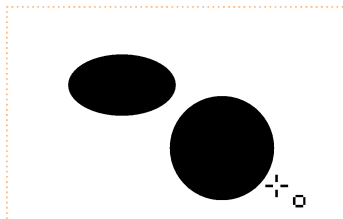


Draw Circle Tool

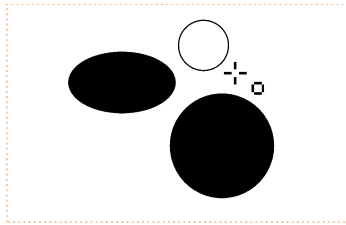
Click the **Draw Circle** tool and drag the pointer () to draw a circle, ellipse, or disk." data-bbox="563 598 592 621"/>



Hold down the **Shift** key to draw circles.



You can draw both filled and unfilled circles, depending on the currently selected fill and edge attributes.

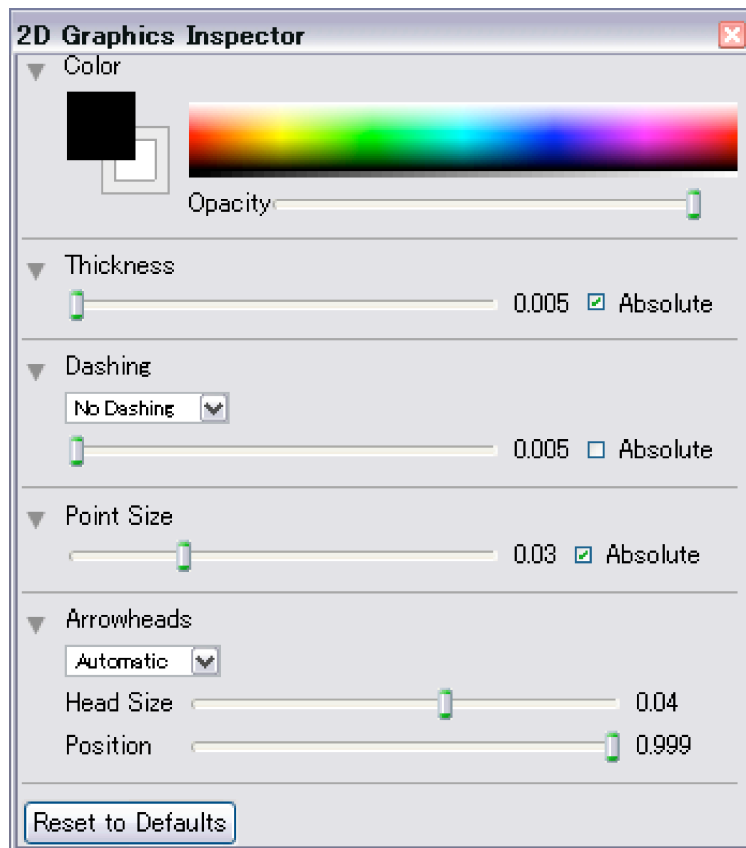


Graphics Inspector



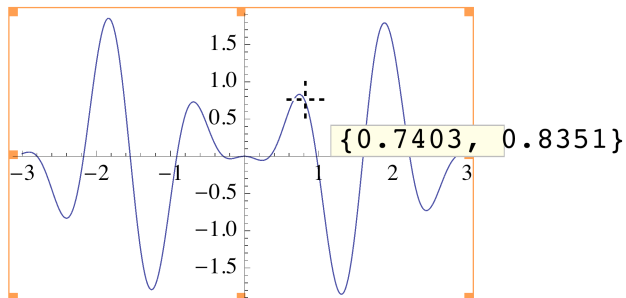
Click the **Graphics Inspector** button to display the **Graphics Inspector** palette.

The **Graphics Inspector** palette lets you interactively set the style of a graphics object.

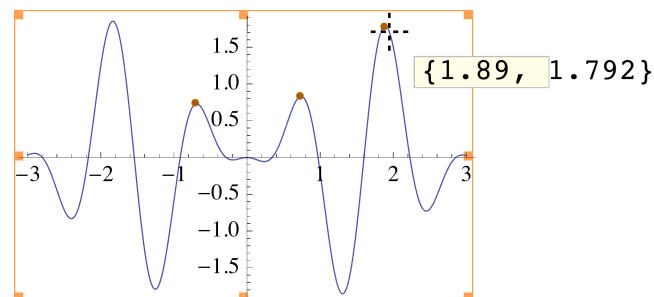


Get Coordinates Tool

Click the **Get Coordinates** tool and move the mouse pointer () over a 2D graphic or a 2D plot. The approximate coordinate values of the mouse position are displayed.



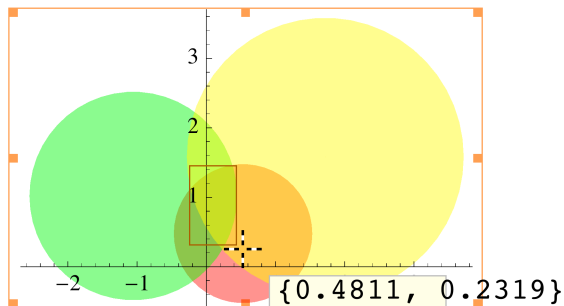
Click to mark the coordinates. Click at other positions to add markers. You can delete markers with **Command + click**.



Use **Command + C** to copy the marked coordinates to the clipboard. Use **Command + V** to paste the copied coordinate values into an input cell.

`{{1.89, 1.792}, {0.7403, 0.8351}, {-0.6706, 0.7132}}`

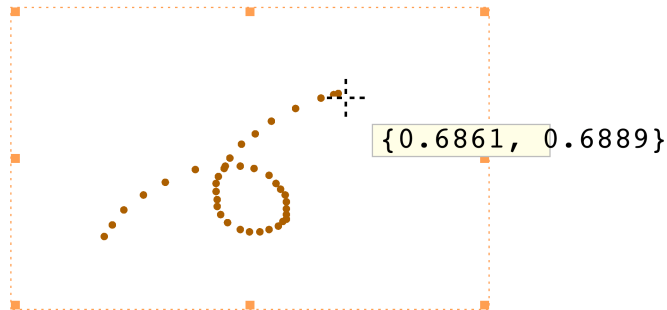
Use **Alt + drag** to mark a rectangle and **Command + C** to copy the coordinates of the opposite corners.



Use Command+V to paste.

```
{{{-0.2292, 0.2501}, {0.4628, 1.452}}}
```

Drag to mark a coordinate path, then type Command+C to copy the coordinates.



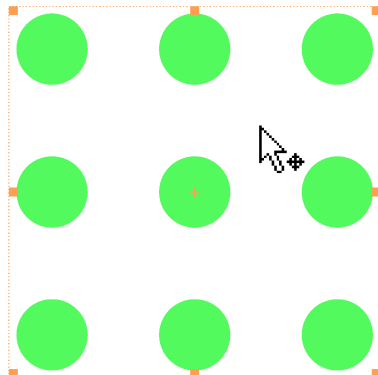
Paste the coordinates into an input cell.

```
{{{0.1222, 0.3444}, {0.1417, 0.3722}, {0.1694, 0.4083},  
{0.2167, 0.4444}, {0.2694, 0.475}, {0.3417, 0.5056}, {0.4139, 0.5139},  
{0.45, 0.5139}, {0.4833, 0.5083}, {0.5194, 0.4917}, {0.5361, 0.4722},  
{0.5472, 0.4583}, {0.5583, 0.4444}, {0.5611, 0.4278}, {0.5611, 0.4111},  
{0.5611, 0.3972}, {0.5611, 0.3861}, {0.5528, 0.3806}, {0.5417, 0.3694},  
{0.5194, 0.3611}, {0.4972, 0.3556}, {0.4722, 0.3556}, {0.45, 0.3611},  
{0.4194, 0.3778}, {0.4028, 0.3972}, {0.3944, 0.4167}, {0.3944, 0.4333},  
{0.3917, 0.4528}, {0.3917, 0.4722}, {0.3972, 0.4889}, {0.4111, 0.5083},  
{0.425, 0.5333}, {0.4528, 0.5667}, {0.4861, 0.5917}, {0.525, 0.6222},  
{0.5833, 0.6528}, {0.6444, 0.6778}, {0.675, 0.6861}, {0.6861, 0.6889}}}
```

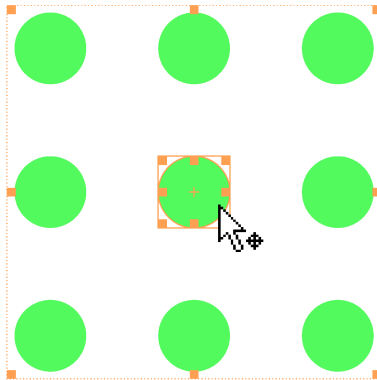
Selecting Graphics Objects

Selecting an Object

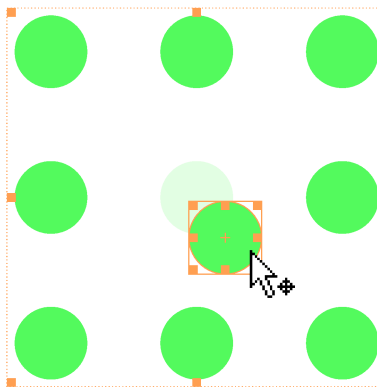
To begin editing within a graphic, double-click the background or an object in the graphic. The frame highlight is dashed.



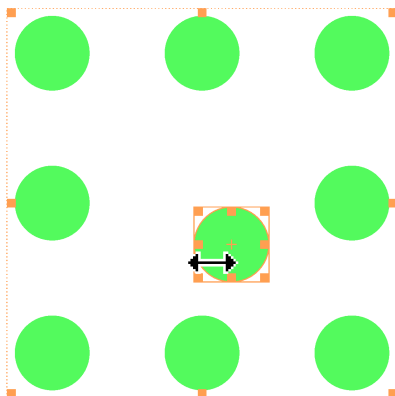
Click a disk to select it.



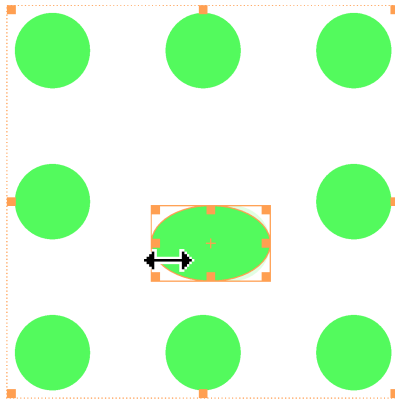
Drag the disk or its frame highlight to move it.



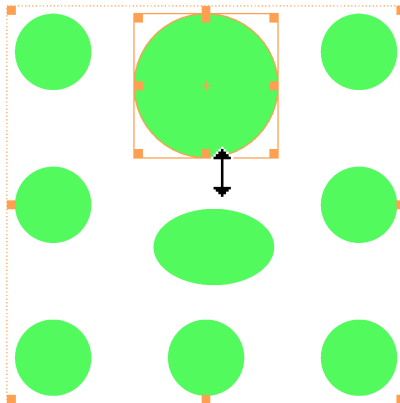
When over a handle, the arrow cursor () changes to the double arrow cursor ( ,  ,  , or ).



Drag a handle to reshape the selected disk.

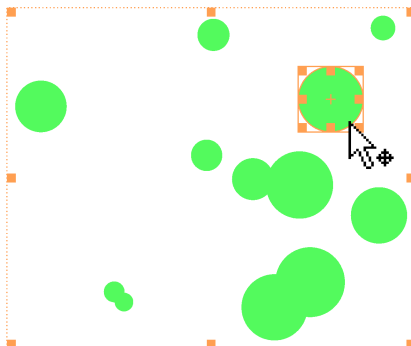


To resize a disk without changing its shape, press **Shift** and drag a handle.

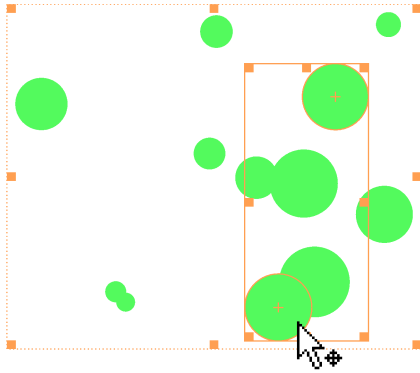


Selecting Multiple Objects

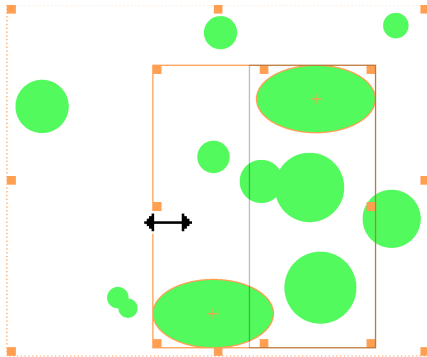
Click a disk to select it.



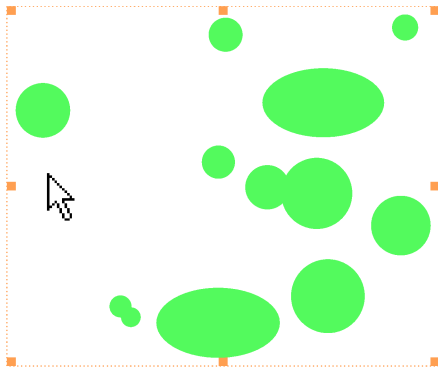
Shift+click to select another disk. The two disks with crosshairs (+) at their centers are selected.



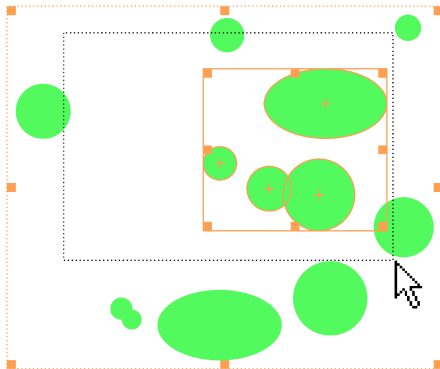
Drag a handle to stretch the selected disks.



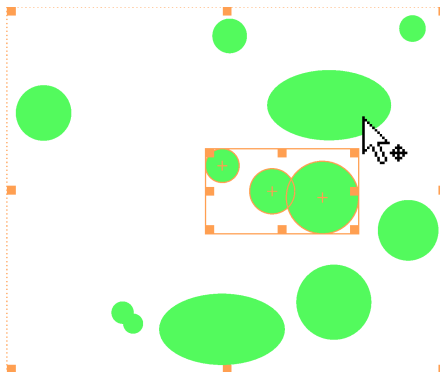
Click outside the selected disks to deselect.



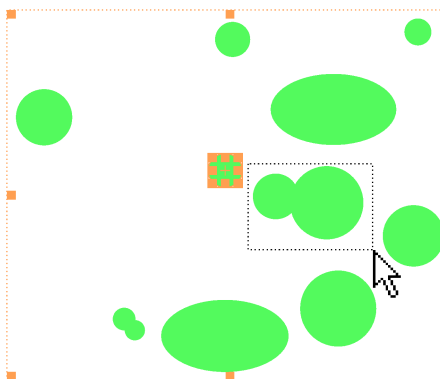
Click and drag across some disks to select them.



Shift+click to deselect a selected disk.



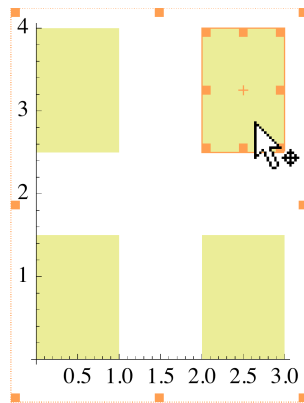
Shift+drag to deselect a group of disks.



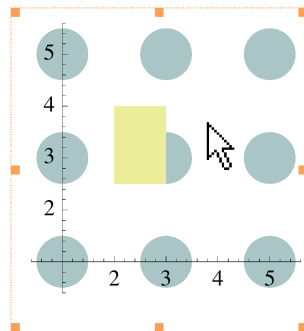
Copy and Paste

The following sequence shows how to copy and paste an object from one graphic to another.

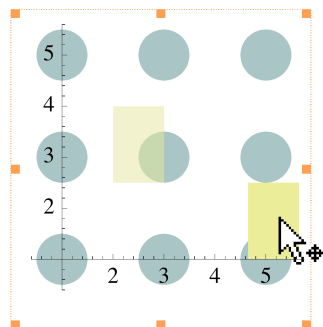
Click to copy a rectangle.



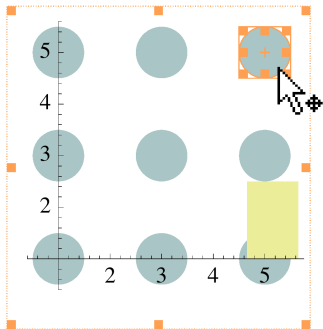
In another graphic, double-click the background and paste. The rectangle will be at its original coordinates.



Drag the pasted rectangle to move it out of the way.

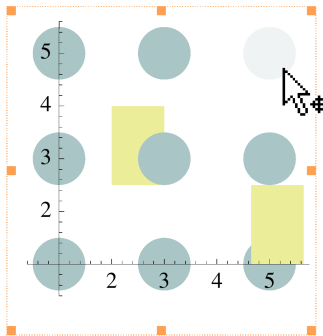


Click a disk to select it.

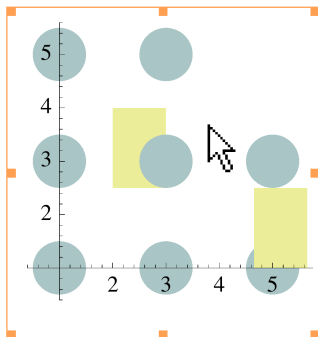


Paste the copied rectangle again. The disk disappears and a new copy of the rectangle appears once again at its original coordinates.

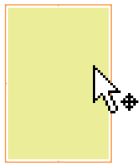
Whether the rectangle appears above or below a disk depends on the position of the replaced disk in the internal ordering of the graphics expression.



In contrast to double-clicking the background of the target graphic, a single click selects the target as a whole.

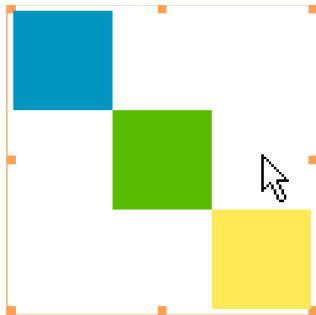


Pasting now replaces the whole graphic.

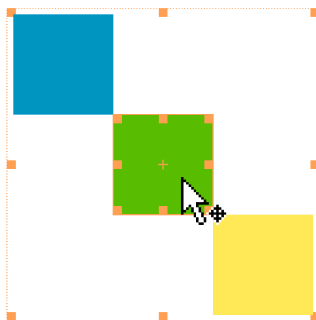


Inset Objects

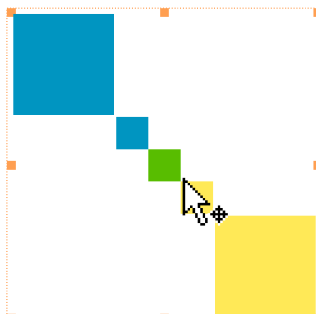
Here is a graphic with three squares. Click the graphic and copy it.



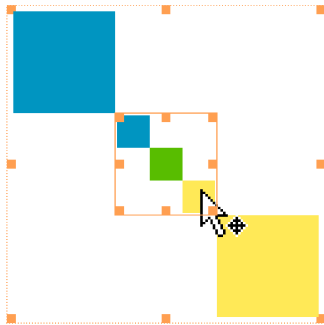
Double-click a square to select it.



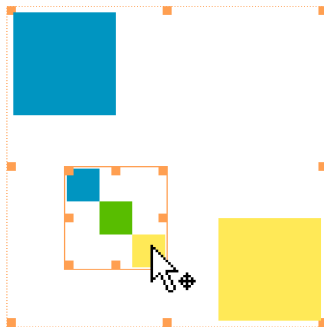
Paste the graphic copied before. The three squares will be pasted, replacing the selected square.



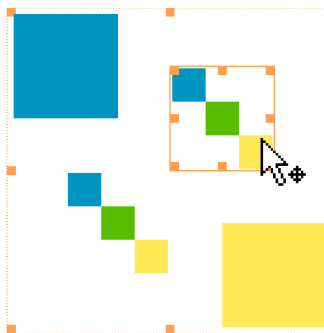
The pasted squares are in an Inset. Click the Inset to select it.



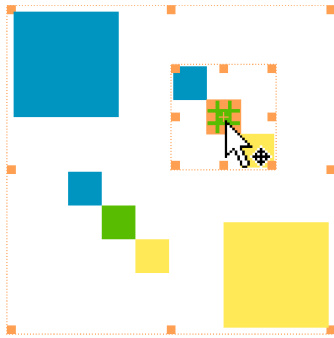
To move the Inset, click anywhere inside it and drag.



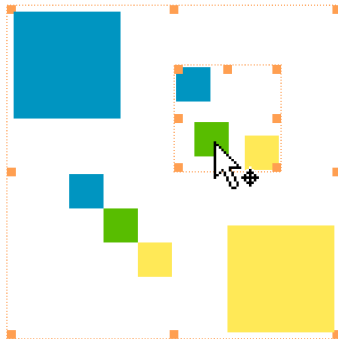
Copy the selected Inset, click the background of the graphic to deselect the Inset, and paste. Move the pasted Inset to the upper right.



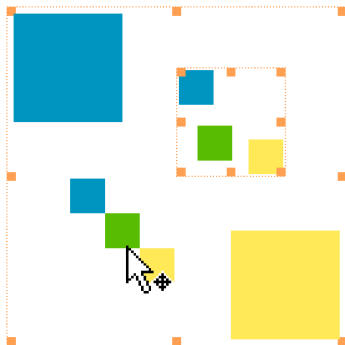
Double-click an object in an Inset to select it.



Click and drag an object in an Inset to move it.



Objects in different Inset groups cannot be selected simultaneously.



Reshaping Graphics Objects

In this tutorial, the following topics are discussed:

Pointers

Vertices and Circle Points

Line Segments

Primitives

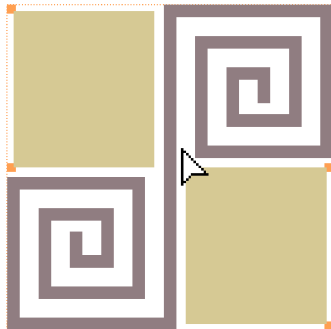
Reshaping Overlapping Objects

Multiple Objects

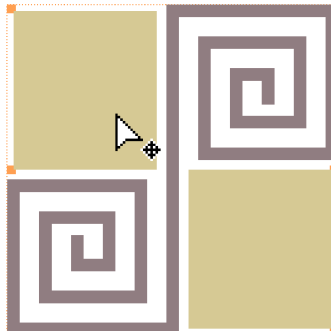
Pointers

The following sequence explains how to use the **Reshape** tool (, , and ).

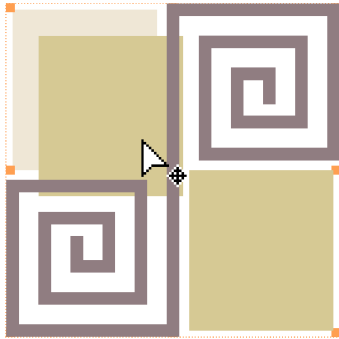
Double-click the background and press the **r** key to make the **Reshape** tool () appear.



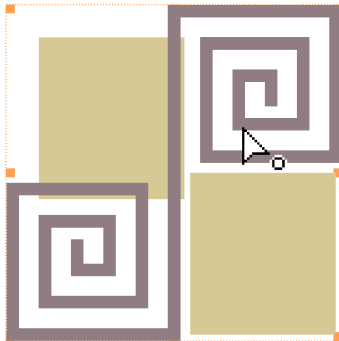
Move the pointer over a filled rectangle—the subscript on the pointer is a pair of double arrows () indicating that the rectangle can be dragged.



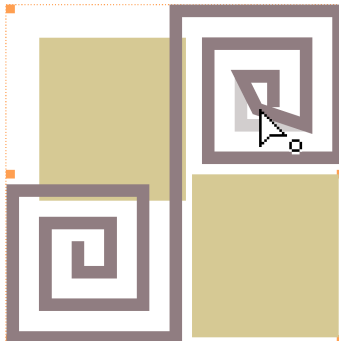
You can now drag the rectangle.



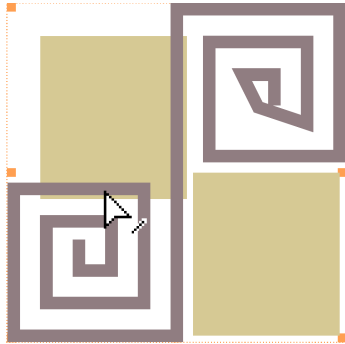
Move the pointer over a vertex—the subscript on the pointer is a small circle ().



Now you can drag the vertex to reshape the line.

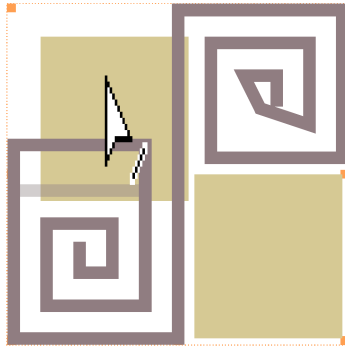


Move the pointer over a line segment—the subscript on the pointer is a small line ().



You can drag the line segment to reshape the line.

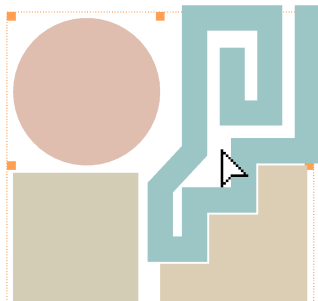
Rectangle primitives do not have selectable line segments.



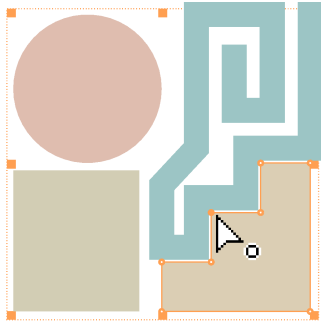
Vertices and Circle Points

The following sequence shows where the selectable points of graphics primitives are located and how to select them.

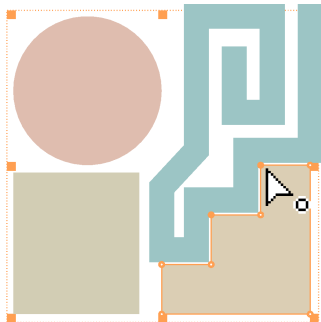
Double-click the background and press the **r** key to make the **Reshape** Tool () appear.



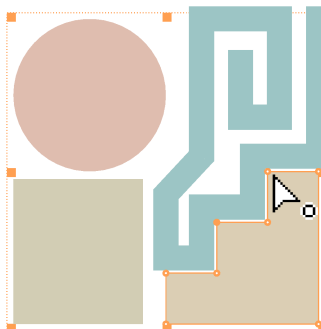
Move the pointer over a vertex—the subscript on the pointer is a small circle (). Click to select the vertex. The other vertices are shown but are not selected.



Shift+click another vertex to add it to the selection.



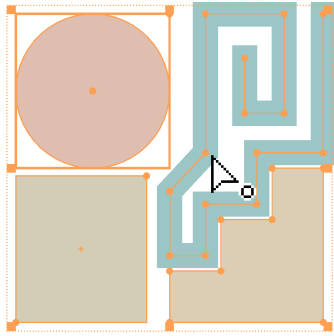
Shift+click a selected vertex to deselect it.



Drag the center or the top-right corner point of a Circle or Disk primitive to reshape it.

The two vertices of a Rectangle primitive are at opposite corners.

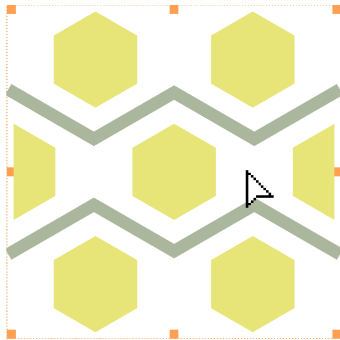
Polygon and Line primitives have vertices at every corner.



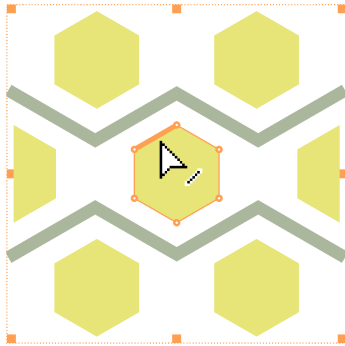
Line Segments

The following sequence shows where the line segments of graphics primitives are located and how to select them.

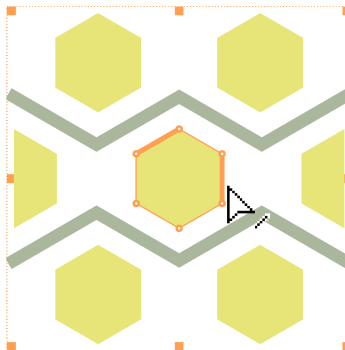
Double-click the background and press the **r** key to make the **Reshape** Tool () appear.



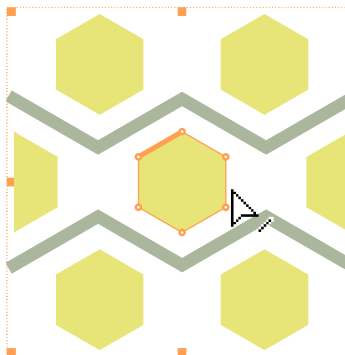
Move the pointer over a line segment—the subscript on the pointer is a small line (). Click to select the line segment.



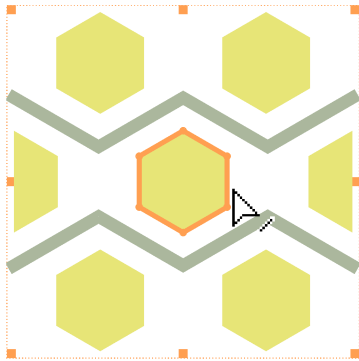
Shift+click another line segment to add it to the selection.



Shift+click a selected line segment to deselect it.

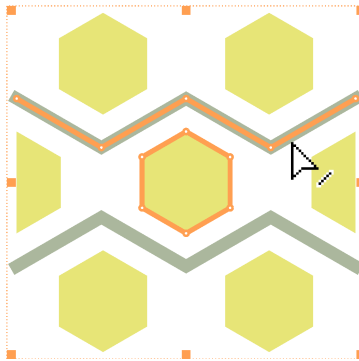


Double-click a line segment to select all the line segments of the polygon.



Polygon and Line primitives have line segments on their sides.

Circle, Disk, and Rectangle primitives do not have selectable line segments.

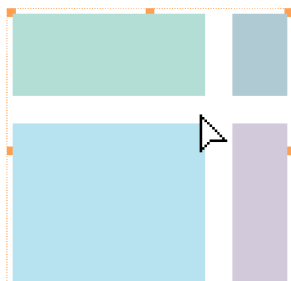


Primitives

Reshaping a Rectangle

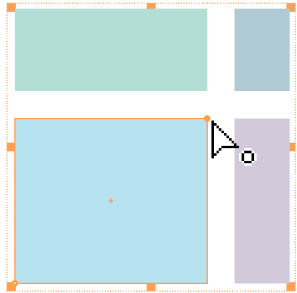
The following sequence shows how to reshape a Rectangle primitive.

Double-click the background and press the **r** key to make the **Reshape Tool** () appear.



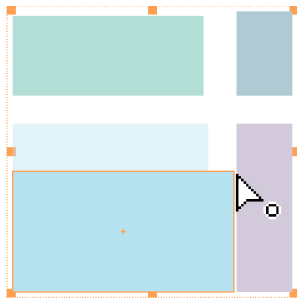
Rectangles have vertices at opposite corners.

Click a vertex to select it.



Drag the selected vertex to reshape the Rectangle.

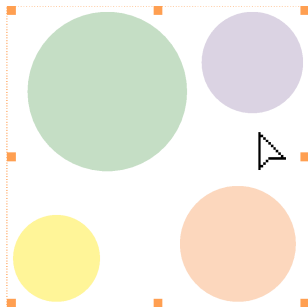
Rectangle primitives do not have line segments that can be selected.



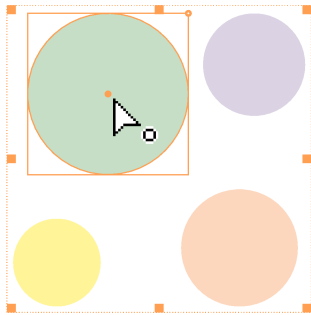
Reshaping a Disk

The following sequence shows how to reshape a Disk primitive.

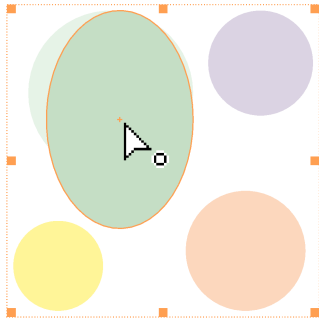
Double-click the background and press the **r** key to make the **Reshape** Tool () appear.



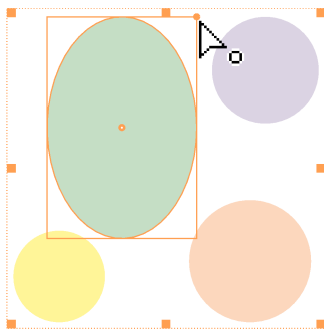
Click the center point of a disk. You can now see the top-right corner point.



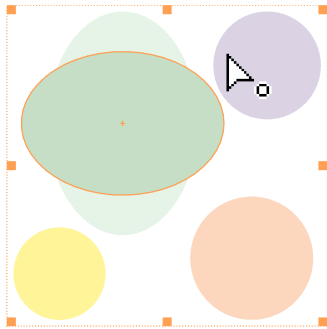
Drag the center point. This keeps the top-right corner point fixed in the same place but changes the shape of the disk.



Click the top-right corner point to select it. This deselects the center point.



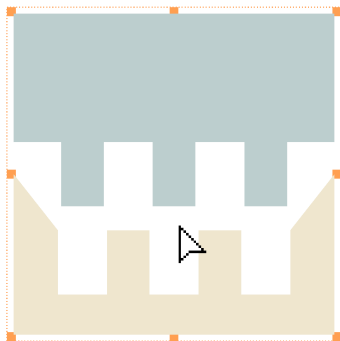
Drag the top-right corner point to change the radius of the disk. The center does not change.



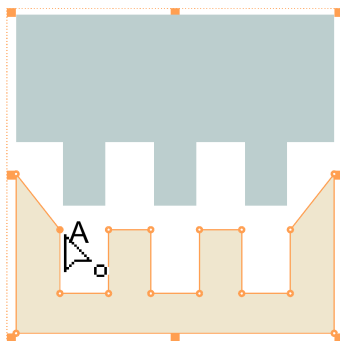
Reshaping a Polygon

The following sequence shows how to reshape the lower polygon to fit the upper polygon.

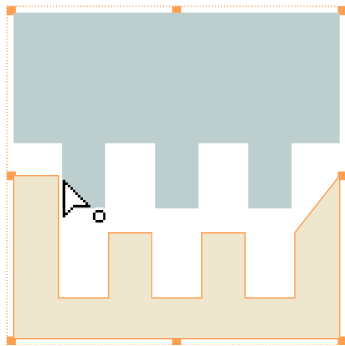
Double-click the background and press the **r** key to make the **Reshape** Tool () appear.



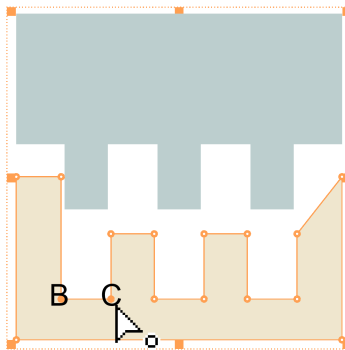
Click the vertex A.



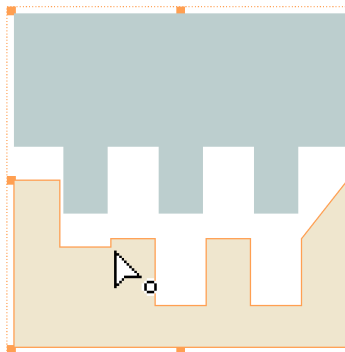
Drag the vertex A to reshape the polygon.



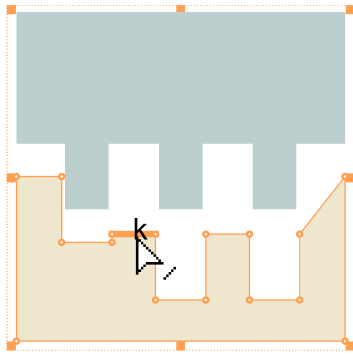
Click the vertex B and **Shift**+click the vertex C.



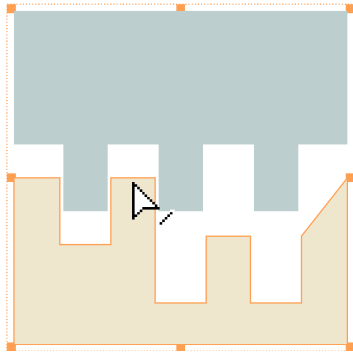
Drag either B or C.



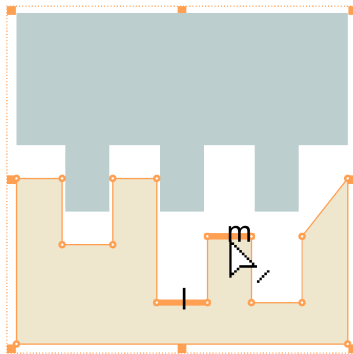
Click the line segment k to select it.



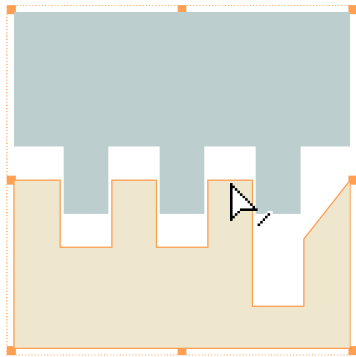
Drag k to reshape the polygon.



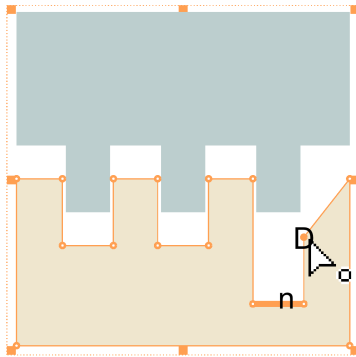
Click the line segment l and Shift+click the line segment m.



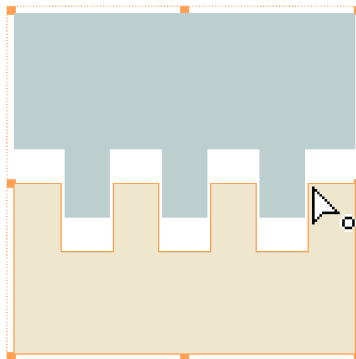
Drag either l or m to move them both together.



Click the line segment n and Shift+click the vertex D.



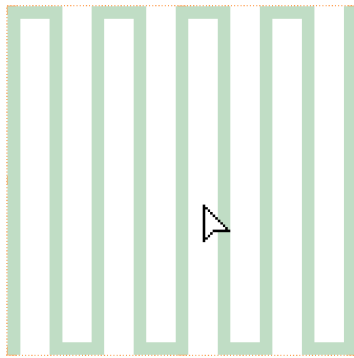
Drag either n or D to move them both together.



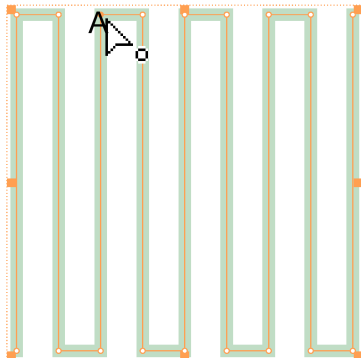
Reshaping Lines

The following sequence shows how to reshape the line below to a zigzag pattern.

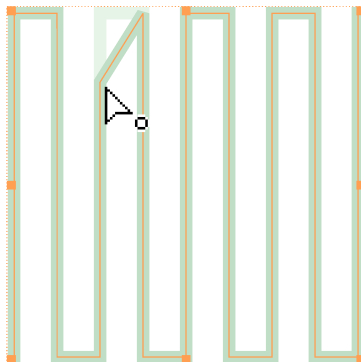
Double-click the background and press the **r** key to make the **Reshape Tool** () appear.



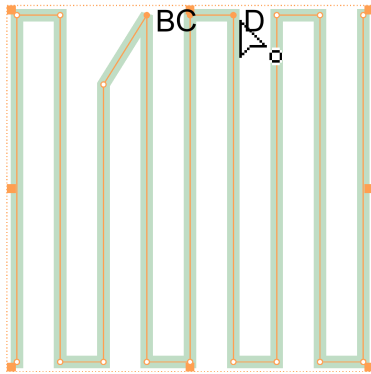
Click the vertex A.



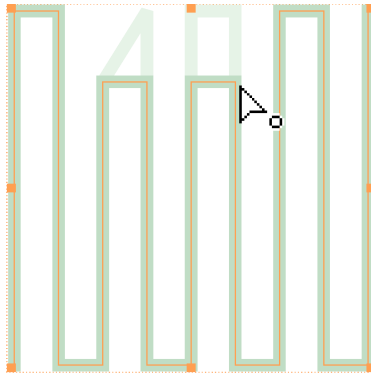
Drag A to reshape the line.



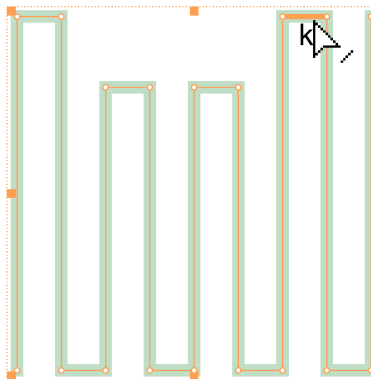
Click the vertex B and **Shift**+click the vertices C and D.



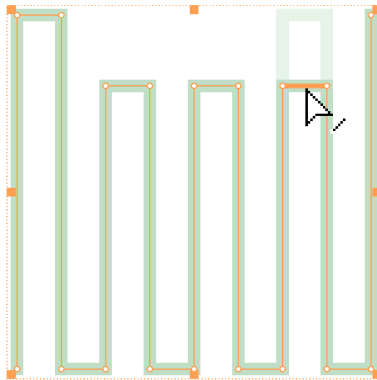
Drag one of the selected vertices.



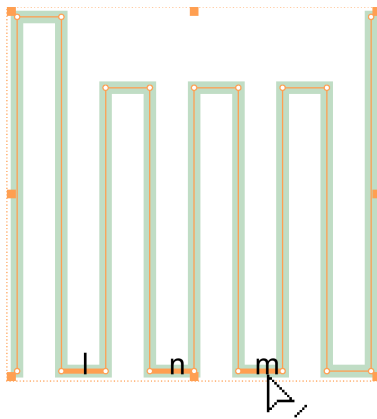
Click the line segment k to select it.



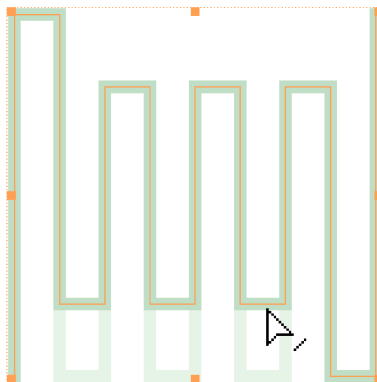
Drag k to reshape the line.



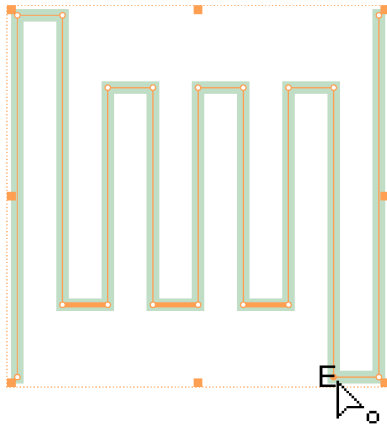
Click the line segment l and **Shift**+click the line segments m and n.



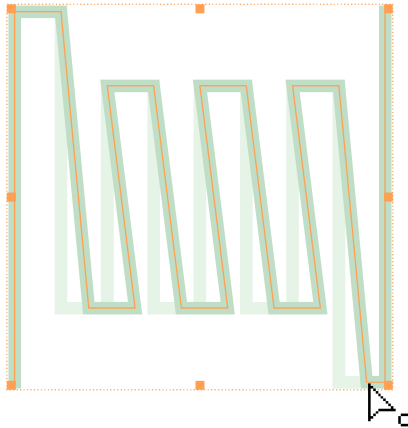
Drag one of the selected line segments to move them all simultaneously.



Shift+click the vertex E to add it to the selection.



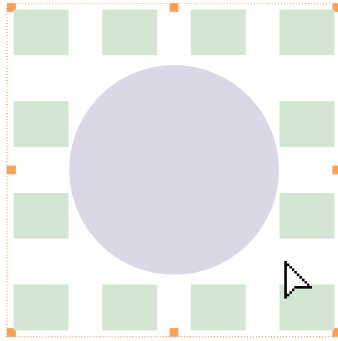
Drag one of the selected line segments or E to move them all simultaneously.



Reshaping Overlapping Objects

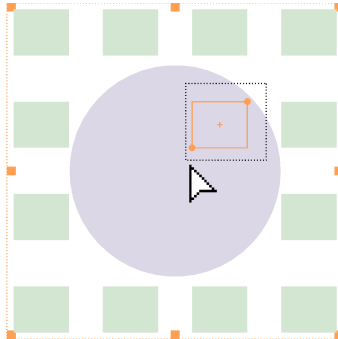
The following sequence shows how to reshape a hidden object.

There are four hidden rectangles underneath the disk in the middle. Double-click the background and press the **r** key to make the **Reshape** Tool () appear.

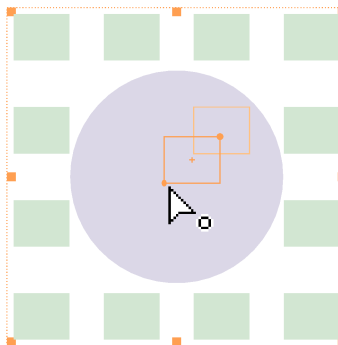


To select one of the hidden rectangles, click the background and drag across the disk.

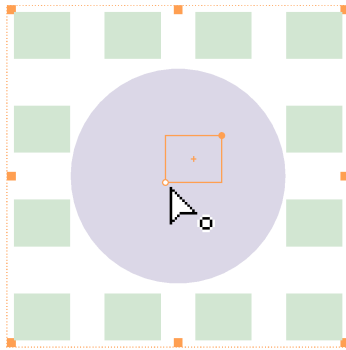
To select the whole rectangle, you may need to start from a new position.



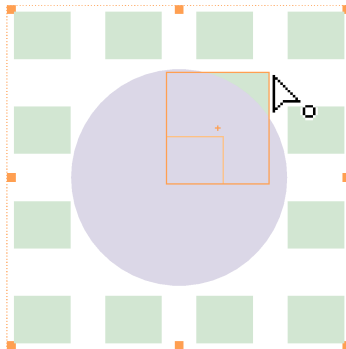
You can move the rectangle by dragging its highlight.



To reshape the selected hidden rectangle, first **Shift**+click a corner to deselect the vertex.



Then drag the other vertex.

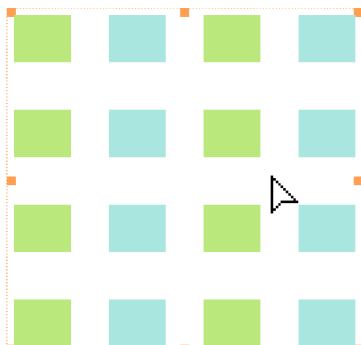


Multiple Objects

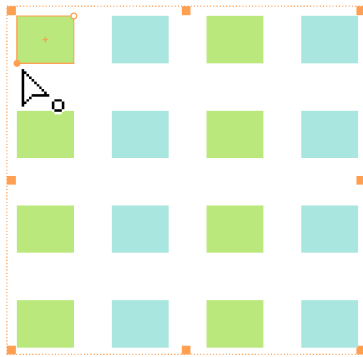
Reshaping Multiple Rectangles

The following sequence shows how to reshape a set of rectangles simultaneously.

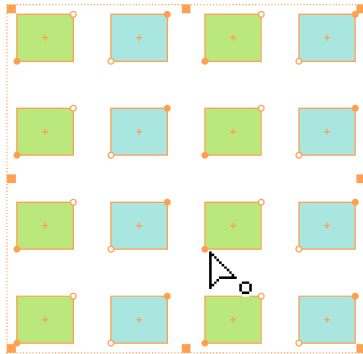
Double-click the background and press the **r** key to make the **Reshape Tool** () appear.



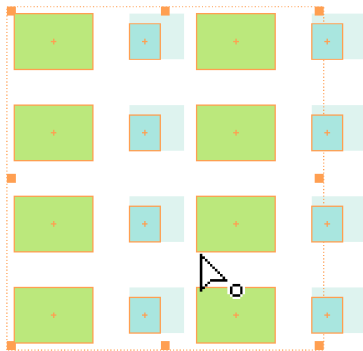
Click one of the two vertices of a rectangle.



Shift+click some vertices in the other rectangles.



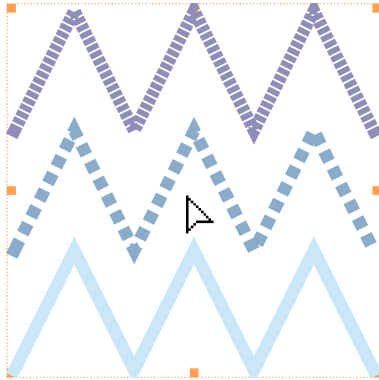
Drag one of the selected vertices. All of the selected rectangles will be reshaped simultaneously.



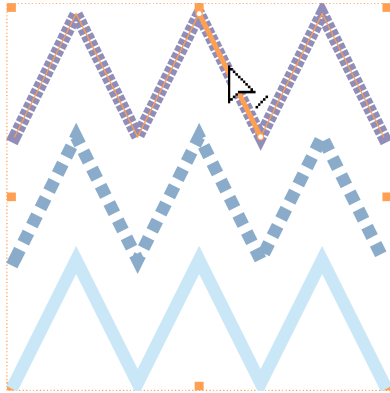
Reshaping Multiple Lines

The following sequence shows how to reshape three lines simultaneously.

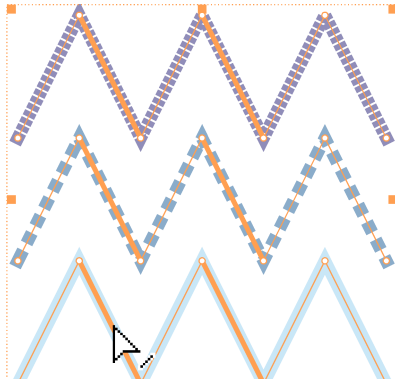
Double-click the background and press the **r** key to make the **Reshape Tool** () appear.



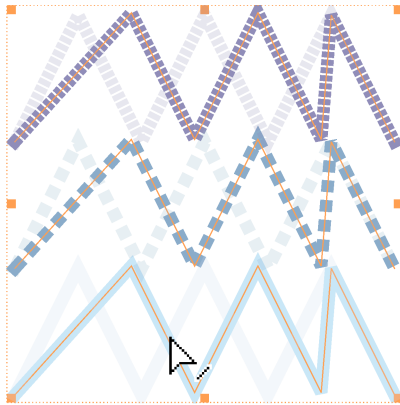
Click a segment of one of the Line primitives.



Shift+click segments in other Line primitives.



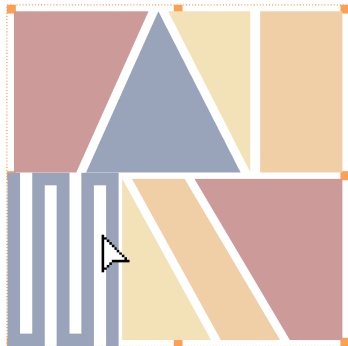
Drag any one of the selected segments to simultaneously reshape the three Line primitives.



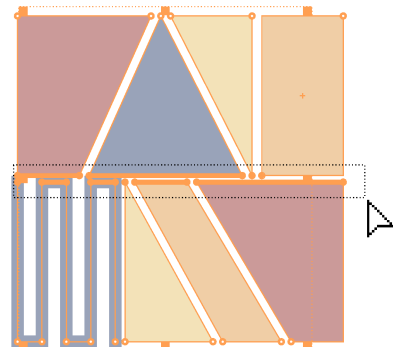
Reshaping with Vertices and Line Segments

The following sequence shows how to reshape with different kinds of objects.

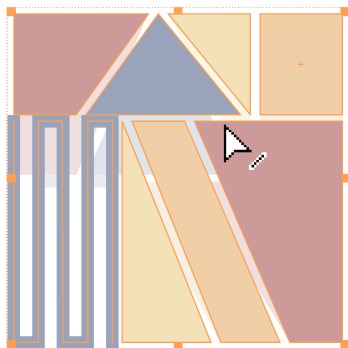
Double-click the background and press the **r** key to make the **Reshape Tool** () appear.



Click the background and drag across all the line segments and vertices around the gap in the middle.



Drag one of the selected line segments or vertices to simultaneously reshape the whole picture.



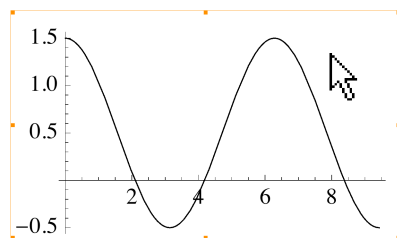
Resizing, Cropping, and Adding Margins to Graphics

Changing the AspectRatio

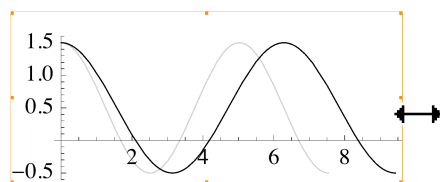
The following sequence shows how to change the aspect ratio of a plot.

Using the **Selection** tool, click to select the graphic.

`Plot[Cos[x] + 1 / 2, {x, 0, 3 Pi}]`



Shift+drag a handle to change the aspect ratio.

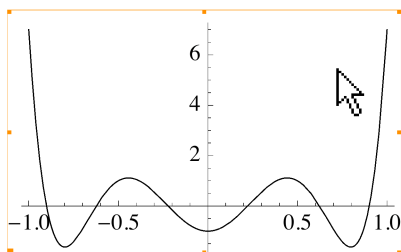


Cropping

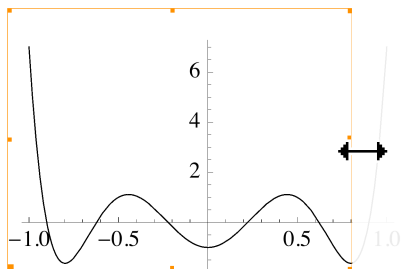
The following sequence shows how to crop a portion of a plot.

Using the **Selection** tool, click a graphic to select it.

`Plot[ChebyshevU[6, x], {x, -1, 1}]`



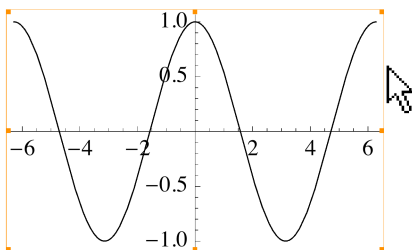
Command +drag a handle to crop the plot.



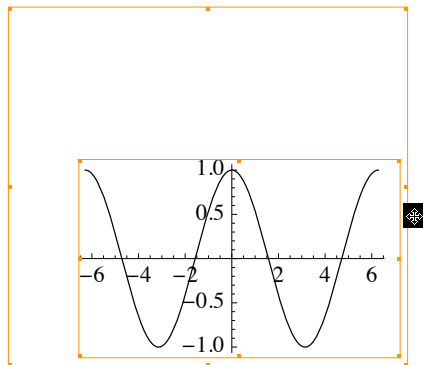
Setting Margins

The following sequence shows how to set the margins of a graphic.

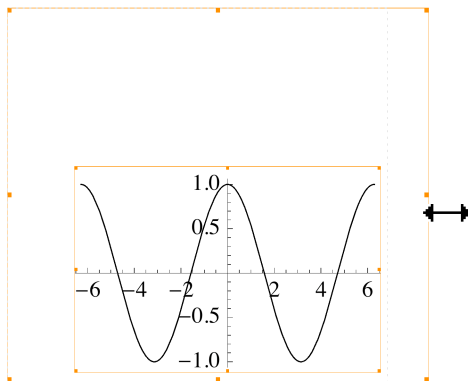
Click to select the graphic.



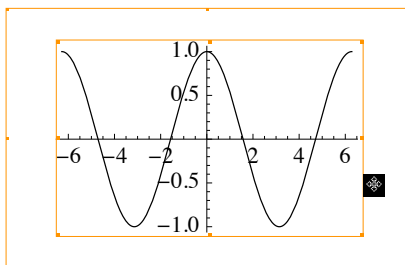
Drag the frame (not a handle) to push out the margins.



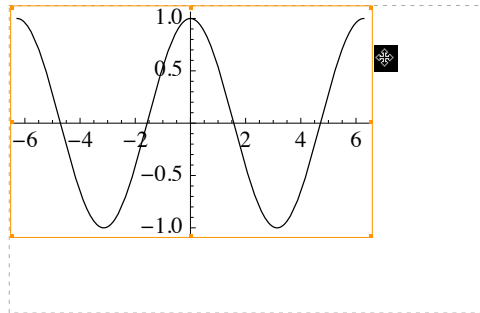
To change a margin directly, drag a margin frame handle.



To keep the margins equal, **Shift**+drag the frame.



To get rid of the margins, **Shift**+drag the graphics frame to the top-left corner.

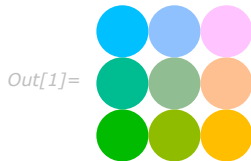


Graphics as Input

An image is equivalent to its symbolic expression. You can operate on an image as you would on a symbolic expression.

This input produces some disks.

```
In[1]:= Graphics[Table[{RGBColor[i / 2, .7, j / 2], Disk[{2 i, 2 j]}], {i, 0, 2}, {j, 0, 2}]]
```



Type `// InputForm` after the graphic to get the symbolic expression that corresponds to the graphic.

```
In[2]:= Graphics[Table[{RGBColor[i / 2, .7, j / 2], Disk[{2 i, 2 j]}], {i, 0, 2}, {j, 0, 2}]]
```



```
Out[3]//InputForm= Graphics[{{RGBColor[0, 0.7, 0], Disk[{0, 0}], {RGBColor[0, 0.7, 1/2],
Disk[{0, 2}], {RGBColor[0, 0.7, 1], Disk[{0, 4]}},
{RGBColor[1/2, 0.7, 0], Disk[{2, 0}], {RGBColor[1/2, 0.7, 1/2],
Disk[{2, 2}], {RGBColor[1/2, 0.7, 1], Disk[{2, 4]}},
{RGBColor[1, 0.7, 0], Disk[{4, 0}], {RGBColor[1, 0.7, 1/2], Disk[{4, 2]}},
{RGBColor[1, 0.7, 1], Disk[{4, 4]}}}]}
```

Type a *Mathematica* Replace command after the output graphic and evaluate.

`In[4]:=`



`/. Disk -> Rectangle`

The disks are now replaced by squares.

`Out[5]=`



Interacting with 3D Graphics

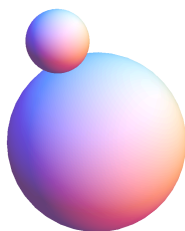
Rotate

You can rotate 3D graphics with your mouse.

Move the pointer over the 3D graphic.

The pointer changes to the 3D rotate pointer .

Drag to rotate the graphic.



Rotate about the Axis Perpendicular to the Screen

You can also rotate 3D objects about the axis perpendicular to the screen with your mouse.

Move the pointer to a corner of the display area of the 3D graphic.

The pointer changes to the 3D vertical rotate pointer .

Click and move the pointer clockwise or counterclockwise.



Zoom In and Out with Command

You can zoom in and out of 3D graphics.

Press **Command** and move the pointer over the 3D graphic.

The pointer changes to the 3D zoom pointer .

Drag up to zoom in and down to zoom out.



Pan with Shift

You can pan 3D graphics across the screen.

Press **Shift** and move the pointer over the 3D graphic.

The pointer changes to the 3D pan pointer .

Drag to move the graphic.

