Reprint from
The *Mathematica* Conference
June, 1992 Boston, MA

877.21.10.1992

Chapter 4 Advanced Graphics

This chapter describes more advanced graphics topics. The first section looks at primitives and directives for two- and three-dimensional graphics. We only look at some of the more complex primitives and directives and shall not review each of them. The following section discusses how to work with color and lighting in *Mathematica* graphics. The final section shows how to combine and convert pictures.

4.1 Primitives and Directives

In previous chapters we have seen how pictures can be made with the built-in functions. We have also seen how the picture which results is a side effect of the evaluation and that there is always a *Mathematica* expression which is returned. This expression contains a *Mathematica* representation of the image. If we have a little knowledge of the nature of this representation we can write our own code to generate similar graphics objects, or take existing objects and adapt them in some way. In this section we shall look a little at some of the components of the *Mathematica* expressions which can be used by graphics commands.

■ 4.1.1 Two-Dimensional Primitives and Directives

Primitives

`Circle`	`Disk`
`Line`	`Point`
`Polygon`	`PostScript`
`Raster`	`RasterArray`
`Rectangle`	`Text`

Two-dimensional graphics primitives.

We can just build expressions containing graphics primitives. The types of coordinates which can be used are discussed in detail in the next chapter. Here the coordinates have their obvious meaning.

This builds a *Mathematica* expression using the graphics primitive `Line`.

```
In[1]:= Line[ {{0,0},{1,0},{1,1}}]

Out[1]= Line[{{0, 0}, {1, 0}, {1, 1}}]
```

The head of `Graphics` is necessary to label the object as containing two-dimensional primitives.

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

Out[2]= -Graphics-
```

The first argument of `Show` must be a graphics object.

```
In[3]:= Show[ Graphics[ Line[ {{0,0},{1,0},{1,1}}]]]
```

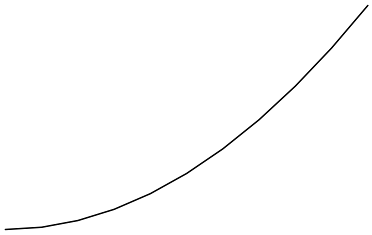
We do not have to explicitly enter numbers for a primitive. So long as we enter an expression which will evaluate to a valid primitive everything works.

```
In[4]:= Line[ Table[ {i, i^2}, {i,0,10}]]

Out[4]= Line[{{0, 0}, {1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25},
    {6, 36}, {7, 49}, {8, 64}, {9, 81}, {10, 100}}]
```

This plots the primitive we just built.

```
In[5]:= Show[ Graphics[ %]]
```



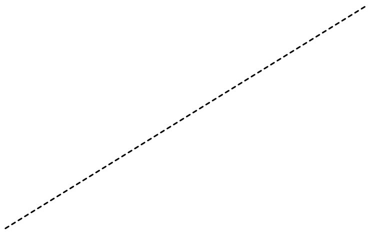
Directives

AbsoluteDashing	AbsolutePointSize
AbsoluteThickness	CMYKColor
Dashing	GrayLevel
Hue	RGBColor
PointSize	Thickness

Two-dimensional graphics directives.

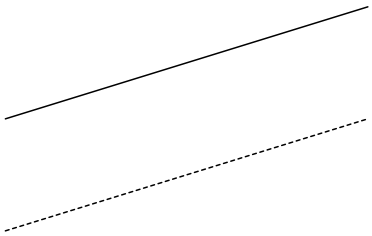
The directive `Dashing` changes the rendering of the `Line` primitive which follows it.

```
In[6]:= Show[ Graphics[ { Dashing[{0.01}],Line[ {{0,0},{1,1}}]}]]
```



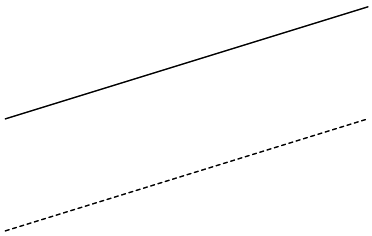
The directive only affects primitives which follow it.

```
In[7]:= Show[ Graphics[
           { Line[ {{ 0,1},{ 1,2}}],
            Dashing[{0.01}],
            Line[ {{0,0},{1,1}}]
            }]]
```



The directive only affects primitives at that level or below. It does not travel back up the tree-structure.

```
In[8]:= Show[
          Graphics[
            {  {
                 Dashing[{0.01}],
                 Line[ {{0,0},{1,1}}]
               },
               Line[{{0,1}, {1,2}}]
             }]]
```



Since we can build graphics objects from primitives we can easily write programs to construct these with the tools provided by *Mathematica*. This is how *Mathematica* packages generate their pictures. The idea of programming to produce pictures is explored in more detail in Chapter 6.

■ 4.1.2 Text

The use of text in *Mathematica* graphics presents various special problems connected with placement and orientation and also with the font which is used to render it.

`DefaultFont`	option to determine the font used
`Text[` *expr*`, {`*x*`, `*y*`}]`	primitive for text of *expr* at `{`*x*`, `*y*`}`
`Text[` *expr*`, {`*x*`, `*y*`}, {`*xoff*`, `*yoff*`}]`	text of *expr* with offset of `{`*xoff*`, `*yoff*`}`
`Text[` *expr*`, {`*x*`, `*y*`}, {`*xoff*`, `*yoff*`}, {`*xvec*`, `*yvec*`}]`	text of *expr* with direction `{`*xvec*`, `*yvec*`}`
`FontForm[` *expr*`, {"fontname", `*size*`}]`	changes the font used for `Text`

Tools for working with text and fonts.

DefaultFont

All graphical commands possess the option `DefaultFont` which defaults to the global variable `$DefaultFont`.

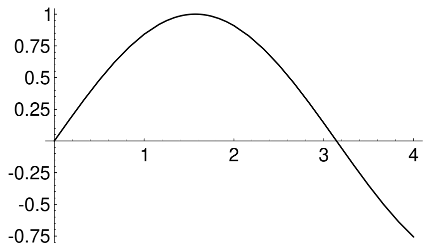
This shows the `$DefaultFont` which was used for this document. Your computer system may have a different value than this.

```
In[9]:= $DefaultFont

Out[9]= {Courier, 5.5}
```

This changes the font used to render text in this plot to 7-point Helvetica.

```
In[10]:= Plot[ Sin[x], {x,0,4}, DefaultFont -> {"Helvetica", 7}]
```



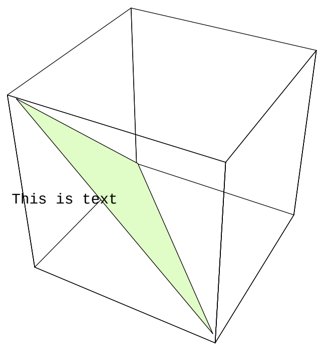
To change the font for all graphics alter `$DefaultFont`. If you are using the kernel renderer which is invoked with `Display` the font can be any PostScript font. It may be that the font you specify is not available to your printer, in which case a default, often Courier, will be substituted.

Text

To include text into graphical images *Mathematica* provides the primitive `Text`. It can be used as a primitive in both `Graphics` and `Graphics3D`.

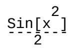
In three dimensions `Text` is always rendered last and is thus never obscured by other graphical objects.

```
In[11]:= Show[ Graphics3D[ {Polygon[{{0,0,1}, {1,0,0}, {0,1,0}}],
                            Text["This is text", {0,.25,.25}] }]]
```



The expression which `Text` takes does not have to be a string.

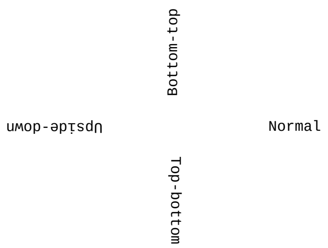
```
In[12]:= Show[ Graphics[ Text[ Sin[x^2]/2, {2,2}]]]
```



The position given to `Text` must be a pair of numbers for `Graphics` and a triple for `Graphics3D`. *Mathematica* will issue a warning message if this is not the case. The offset positions the text in relation to this position and is discussed in The *Mathematica* Book. The default value of the offset, `{0,0}`, centers the string at the given position. The direction is used to rotate the text. It is the vector along which the text will lie.

This shows a variety of rotated `Text` primitives. The option `PlotRange -> All` is given to ensure that all the text is included in the image.

```
In[13]:= Show[ Graphics[{
            Text["Normal", {1,0}, {0,0}, {1,0}],
            Text["Bottom-top", {0,1}, {0,0}, {0,1}],
            Text["Upside-down", {-1,0}, {0,0}, {-1,0}],
            Text["Top-bottom", {0,-1}, {0,0}, {0,-1}]},
            PlotRange -> All]]
```



For some window systems it is very hard to rotate text and consequently it will not appear rotated. When the output is printed to a PostScript printer the text will have has the correct orientation.

FontForm

`FontForm` allows the font for a particular text primitive to be altered from the default font for that graphic.

This changes the font to 15-point Times-Bold.

```
In[14]:= Show[ Graphics[
            Text[  FontForm[ "This is my string",
                   {"Times-Bold", 15}], {0,0}],
                   PlotRange -> All]]
```

■ 4.1.3 Rectangle

`Rectangle[ {`*xmin* , *ymin*`}, {`*xmax* , *ymax*`}]`	simple rectangle
`Rectangle[ {`*xmin* , *ymin*`}, {`*xmax* , *ymax*`}, `*graphic*`]`	rectangle filled with *graphic*

Uses of the `Rectangle` primitive.

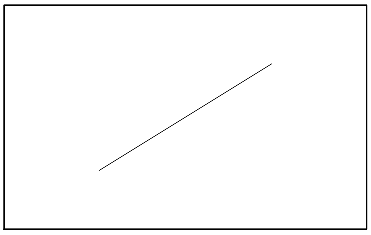
The primitive `Rectangle` if given with two arguments will be colored in the usual way. However a third argument can be given to specify that the `Rectangle` should be filled with a graphics object.

This draws a rectangle filled with gray.

```
In[15]:= Show[ Graphics[ {GrayLevel[ 0.5], Rectangle[{0,0},
          {1,1}]}]]
```

This nests a graphics object inside `Rectangle`.

```
In[16]:= Show[
            Graphics[
              {Line[ {{0,0}, {0,1}, {1,1},
                      {1,0},{0,0}}],
               Rectangle[ {.25, .25}, {.75, .75},
                 Graphics[ Line[ {{0,0}, {1,1}}]]]
               }]]
```



■ 4.1.4 Three-Dimensional Primitives and Directives

`Cuboid`	`Line`
`Point`	`Polygon`
`Text`	

`Graphics3D` primitives.

`AbsoluteDashing`	`AbsolutePointSize`
`AbsoluteThickness`	`CMYKColor`
`Dashing`	`EdgeForm`
`FaceForm`	`GrayLevel`
`Hue`	`PointSize`
`RGBColor`	`SurfaceColor`
`Thickness`	

`Graphics3D` directives.

The primitives and directives for `Graphics3D` work in a fashion analogous to `Graphics`. The primitives take triples of numbers instead of pairs. The specification of coordinates is described in the next chapter.

FaceForm and EdgeForm

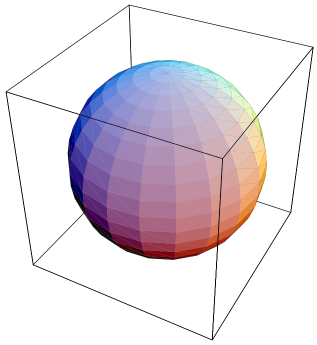
`FaceForm` and `EdgeForm` alter just the edge or the face of a polygon. In addition, `FaceForm` allows the front and back of the polygon to be described differently.

`EdgeForm[]`	do not draw polygons edges
`EdgeForm[` *dir* `]`	use *dir* to draw polygons edges
`FaceForm[` *dir* `]`	draw the polygon faces with *dir*
`FaceForm[` *dirf*`,` *dirb* `]`	draw the front face with *dirf* and back face with *dirb*

```
In[17]:= prims = First[
          ParametricPlot3D[
          {Sin[t] Cos[p] ,
           Sin[t] Sin[p], Cos[t]},
          {t,0,Pi}, {p, 0, 2Pi},
          DisplayFunction -> Identity]];
```

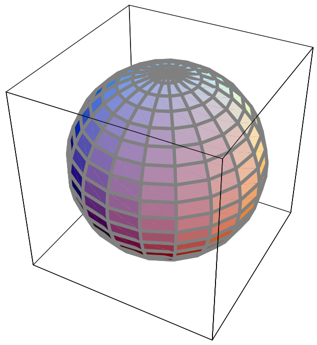
This does not draw lines at the polygon edges.

```
In[18]:= Show[ Graphics3D[ { EdgeForm[ ], prims}]]
```



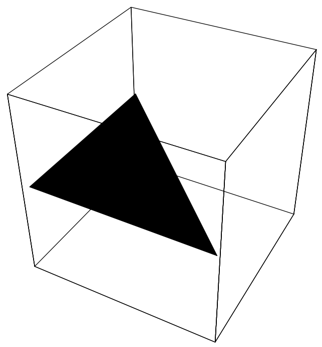
This makes them thick and gray.

```
In[19]:= Show[ Graphics3D[ {
          EdgeForm[ {GrayLevel[0.5], Thickness[0.01]}],
          prims}]]
```



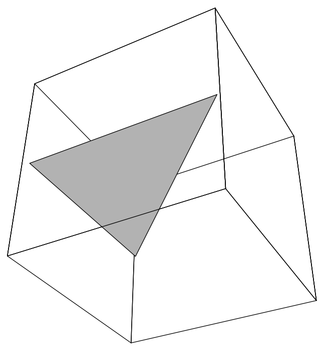
This shows the top view of the polygon.

```
In[20]:= Show[
           Graphics3D[
             {FaceForm[ GrayLevel[ 0], GrayLevel[0.7]],
              Polygon[{{0,0,0},{1,0,0},{0,1,0}}]}],
           Lighting -> False]
```



This shows the bottom, which has a different color.

```
In[21]:= Show[ %, ViewPoint -> {1.3, -2.4, -2.}]
```



SurfaceColor

The `SurfaceColor` directive allows one to construct more complex surface properties. It defines the diffuse and specular reflectivities of polygons.

`SurfaceColor[` *difcol* `]`	diffuse reflector of color *difcol*
`SurfaceColor[` *difcol*`,` *specol*`]`	specular reflector of color *specol*
`SurfaceColor[` *difcol*`,` *specol*`,` *exp*`]`	specular exponent *exp*

4.2 Using Color and Lighting

Mathematica graphics have many features designed to use color. Some of these examples will not look so interesting on a monochrome display; however, in color some of them are quite striking.

- Directives
- Lighting
- Function evaluation
- Color output

Some of the ways to introduce color into *Mathematica* graphics.

4.2.1 Directives

There are a number of directives to color primitives.

```
GrayLevel[g]
RGBColor[r, g, b]
Hue[h]
Hue[h, s, b]
CMYKColor[c, m, y, k]
```

Mathematica color directives.

The directives `GrayLevel`, `RGBColor` and `CMYKColor` take as arguments numbers between 0 and 1 inclusive. Thus `GrayLevel[0]` is black, `RGBColor[1,0,0]` is a red and `CMYKColor[0,1,0,0]` is a magenta.

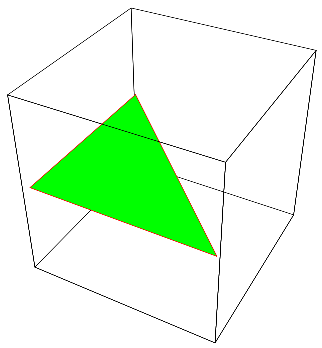
The directive `Hue` is special; it can take one or three arguments. The latter, `Hue[h, s, b]`, describes the hue, saturation and brightness. There is also a one-argument form, `Hue[h]`, which is equivalent to `Hue[h, 1, 1]`. The arguments for saturation and brightness must be numbers between 0 and 1 inclusive. However the first argument, that for hue, is taken modulo 1. Thus `Hue[0]`, `Hue[1]`, and `Hue[2]` are all the same color, namely a red. This wrapping around makes `Hue` a versatile command for introducing color into the graphics.

The directive `GrayLevel` colors the `Rectangle`.

```
In[22]:= Show[ Graphics[ {
              GrayLevel[.5],
              Rectangle[{0,0}, {1,1}]}]]
```

`Lighting -> False` switches off the default coloring which is determined by the light sources. `FaceForm` and `EdgeForm` are useful methods to color polygons.

```
In[23]:= Show[ Graphics3D[ {
                 EdgeForm[RGBColor[1,0,0]],
                 FaceForm[RGBColor[0,1,0]],
                 Polygon[{{0,0,0}, {1,0,0}, {0,1,0}}]},
         Lighting -> False]]
```



4.2.2 Lighting

The lighting of three-dimensional *Mathematica* graphics is set by a number of options.

`Lighting`	whether or not to use simulated lights
`LightSources`	positions and colors of the lights
`AmbientLight`	ambient light color
`Shading`	whether or not to color the surface

Options for controlling lighting and coloring of three-dimensional graphics.

The lighting model used in *Mathematica* is not a photo-realistic ray tracer. It provides simulated light sources which do not diverge and do not cast shadows. To use the simulated light sources it is necessary to set the option `Lighting -> True`, which is set by default. The directions of the lights are determined by the setting of the `LightSources` options.

The default setting of the light sources.

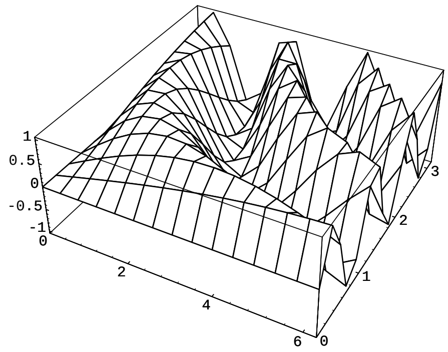
```
In[24]:= Options[ Graphics3D, LightSources]

Out[24]= {LightSources ->
   {{{1., 0., 1.}, RGBColor[1, 0, 0]},
    {{1., 1., 1.}, RGBColor[0, 1, 0]},
    {{0., 1., 1.}, RGBColor[0, 0, 1]}}}
```

The coordinates of these light sources are in display coordinates. Thus the default setting gives lights which are on the right, top and top-right of the screen. The parts of the object which are not illuminated by the light sources are colored by the `AmbientLight` option. To make a more complex type of surface one can use the `SurfaceColor` directive.

`Shading -> False` will give a white surface.

```
In[25]:= Plot3D[ Sin[x y], {x,0,2Pi}, {y,0,Pi}, Shading -> False]
```



4.2.3 Function Evaluation

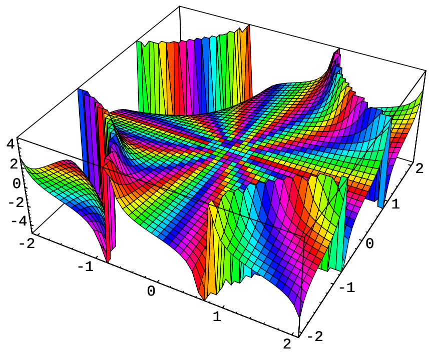
Mathematica provides a number of ways to color a function according to an evaluation.

- `Plot3D[ {`*fun*`[`*x*`, `*y*`], `*color*`[`*x*`, `*y*`]}, ...]`
- `ParametricPlot3D[ {`*x*`, `*y*`, `*z*`, `*color*`[`*x*`,`*y*`,`*z*`]}, ..., Lighting -> False]`
- `Plot3D[ `*fun*`[`*x*` , `*y*`], ...., ColorFunction -> `*col*`]`
- `DensityPlot[ `*fun*`[x , y], ...., ColorFunction -> `*col*`]`
- `ContourPlot[ `*fun*`[`*x*` , `*y*`], ...., ColorFunction -> `*col*`]`

Some ways to color plots by function evaluation.

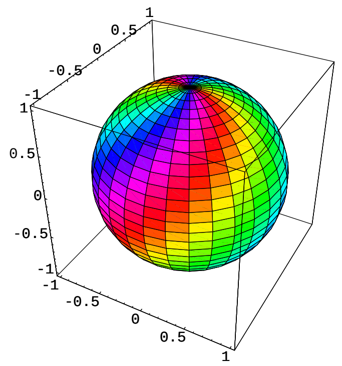
Height is determined by the first element of the list and the color by the second.

```
In[26]:= Plot3D[ {Tan[x y], Hue[ ArcTan[ x,y]/2Pi]},
           {x,-2Pi/3,2Pi/3}, {y,-2Pi/3,2Pi/3},
           PlotPoints -> 50]
```



The fourth coordinate determines the color of the patch.

```
In[27]:= ParametricPlot3D[
             {Cos[p] Sin[t], Sin[p] Sin[t], Cos[t],
             FaceForm[Hue[(t+p)/Pi]]},
             {p,0,2Pi}, {t,0,Pi},
             Lighting -> False, PlotPoints -> 30]
```

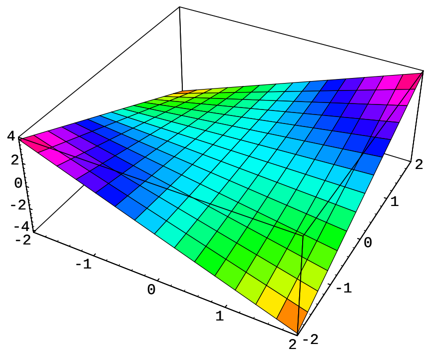


ColorFunction

`ColorFunction` is an option for `SurfaceGraphics`, `DensityGraphics`, `ContourGraphics`, and related commands. It also exists for the `Graphics` primitive `Raster`. It must be set to a function of one argument. This is given the height at each point of the array scaled to run from 0 to 1. It must return a color directive.

Regions which are red will be the very highest or the very lowest since `Hue` wraps around. This will use the one argument version of `Hue`. This is a very simple way to color a plot.

```
In[28]:= Plot3D[ x y, {x,-2,2}, {y,-2,2}, ColorFunction -> Hue]
```

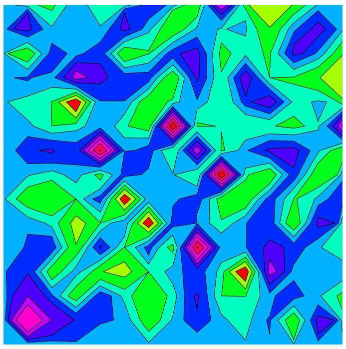


This defines a `ColorFunction`. Note that it is a function which takes one argument.

```
In[29]:= col[ z_] := Hue[ z^0.65]
```

This looks interesting on a color display.

```
In[30]:= ContourPlot[ Evaluate[
                  Sum[ Sin[n x m y], {n,5}, {m,5}]],
            {x,0,2Pi},{y,0,2Pi}, ColorFunction -> col,
            Frame -> None]
```



■ 4.2.4 ColorOutput

`ColorOutput` is an option for all graphics commands. It instructs *Mathematica* to filter through the PostScript as it is emitted and change color specifications. In this way color separations can be carried out.

`Automatic`	leave original colors untouched
`GrayLevel`	convert to `GrayLevel` with internal algorithm
`RGBColor`	convert to `RGBColor` with internal algorithm
`CMYKColor`	convert to `CMYKColor` with internal algorithm
fun	use supplied function to convert colors

Values of the option `ColorOutput`.

The default value of `ColorOutput` is `Automatic`. This has no effect upon colors.

Mathematica will use an internal algorithm to convert PostScript `setrgbcolor` to `setgray`.

```
In[31]:= plot =  Show[ Graphics[{
                 RGBColor[1,0,0],
                 Rectangle[{0,0}, {1,1}]},
           ColorOutput -> GrayLevel]]
```

```
In[32]:= colsep[ RGBColor[ r_, g_, b_]] :=
           CMYKColor[ 1-r, 1-g, 1-b, 0]
```

`colsep` is a simple color separation algorithm.

```
In[33]:= colsep[ GrayLevel[ g_]] :=
           CMYKColor[ 0, 0, 0, g]
```

```
In[34]:= colsep[ RGBColor[ 1,1,0]]

Out[34]= CMYKColor[0, 0, 1, 0]

In[35]:= colsep[ GrayLevel[ 0]]

Out[35]= CMYKColor[0, 0, 0, 0]
```

This uses `colsep`.

```
In[36]:= Show[ plot, ColorOutput -> colsep]
```

4.3 Combining and Converting Images

There are a number of ways to combine and convert images which are possible.

`Prolog`, `Epilog`	include `Graphics` primitives
`Plot`, `ParametricPlot`, `ParametricPlot3D`	combine plots of several functions
`Graphics3D[` *obj*`]`	convert to `Graphics3D` object
`Graphics[` *obj*`]`	convert to `Graphics` object
`Show[` *obj1*`,` *obj2*`, ...]`	convert and combine several objects

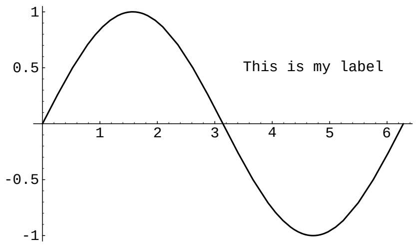
Ways to combine and convert *Mathematica* graphical images.

Prolog and Epilog

`Prolog` and `Epilog` allow one to include `Graphics` primitives in graphical objects of all types. The former is rendered before the picture is drawn and the latter afterwards. This can be used to add extra labels to plots or to nest plots with the `Rectangle` primitive.

This is a useful way to label a plot.

```
In[37]:= Plot[ Sin[x], {x,0,2Pi}, Prolog -> Text["This is my
         label", {3Pi/2, .5}]]
```

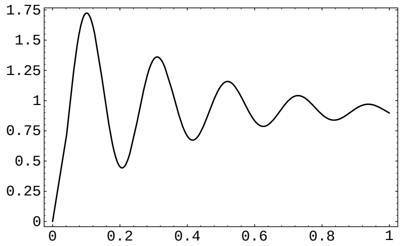


This defines a function `funct` which has a different long-time behavior compared with the short-time behavior. The last section describes *Mathematica* functions in more detail.

```
In[38]:= funct[ x_] := Exp[ -x/10] - Cos[ 30 x] Exp[ -3x]
```

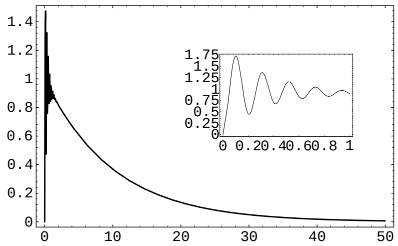
This makes a plot of the short-time behavior and stores it as the symbol `short`.

```
In[39]:= short = Plot[ funct[x], {x,0,1}, Frame -> True]
```



This plots `funct` over a longer time scale but includes a subplot of the short-time behavior. The `Prolog` option allows one to render primitives before the main picture is drawn. It is a useful method to include explicit primitives into a `Plot`. The nested plot can be any *Mathematica* graphic.

```
In[40]:= Plot[ funct[x], {x,0,50},
             Frame -> True,
             Prolog -> Rectangle[ {18,.5}, {48,1.2}, short]]
```

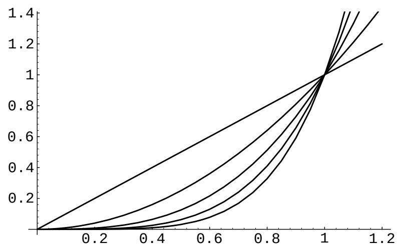


Plotting Several Functions

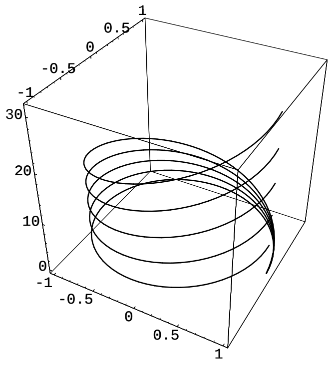
The function plotting commands `Plot`, `ParametricPlot` and `ParametricPlot3D` all allow more than one function to be plotted.

The `Evaluate` command is necessary since `Plot` is `HoldAll`.

```
In[41]:= Plot[ Evaluate[ Table[ x^n, {n,5}]], {x,0,1.2}]
```



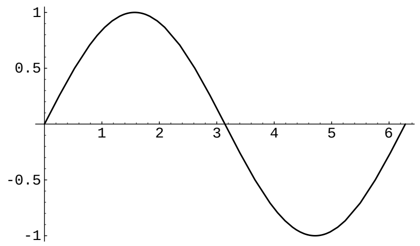
```
In[42]:= ParametricPlot3D[ Evaluate[
                              Table[{ Cos[x], Sin[x], n x},
                                                   {n,5}]],
                              {x,0,2Pi}, BoxRatios -> {1,1,1}]
```



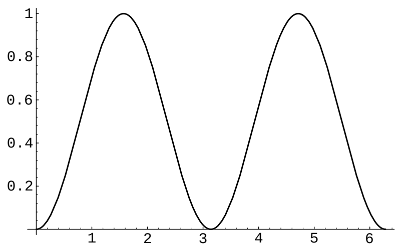
Show

To combine existing images we can just use `Show`.

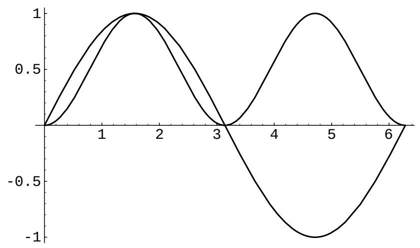
```
In[43]:= Plot[ Sin[x], {x,0,2Pi}]
```



```
In[44]:= Plot[ Sin[x]^2, {x,0,2Pi}]
```



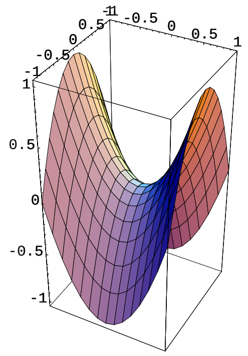
```
In[45]:= Show[%, %%]
```



There are also methods to convert graphics of one type to another. One of the most common is the conversion of `SurfaceGraphics` to `Graphics3D`. This is useful since `Graphics3D` primitives can intersect and generally represent more complex shapes than `SurfaceGraphics`.

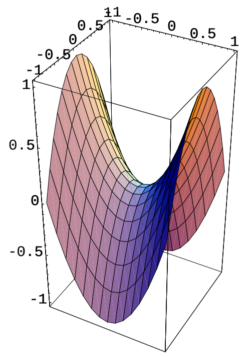
This makes a `SurfaceGraphics` object.

```
In[46]:= plot = Plot3D[ x^2 - y^2 , {x,-1,1}, {y,-1,1},
                 BoxRatios -> {1,1,2}]
```



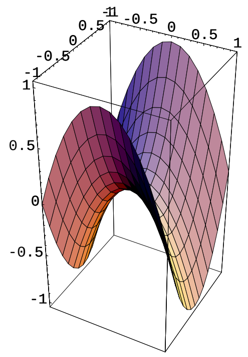
This converts it to `Graphics3D`.

```
In[47]:= Show[ Graphics3D[ %]]
```



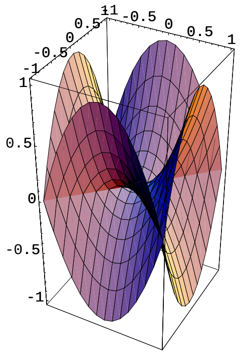
This makes another `SurfaceGraphics` object.

```
In[48]:= plot1 = Plot3D[ y^2 - x^2,  {x,-1,1}, {y,-1,1},
                  BoxRatios -> {1,1,2}]
```



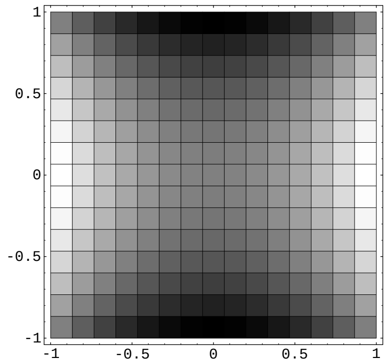
The conversion to `Graphics3D` happens automatically.

```
In[49]:= Show[plot1,plot]
```



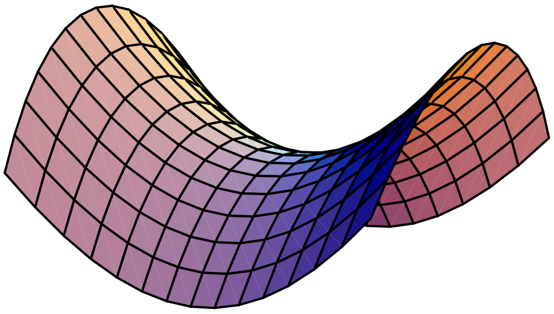
One can convert to other graphics forms as well.

```
In[50]:= Show[ DensityGraphics[ plot]]
```



This converts to a `Graphics` object. Note that since only primitives are converted the axes are lost.

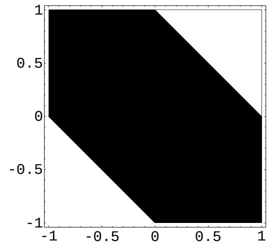
```
In[51]:= Show[ Graphics[ plot]]
```



Another useful feature which is provided in Version 2.1 *Mathematica* is the conversion of `ContourGraphics` objects to `Graphics`. Thus the list of `Graphics` primitives which draw the contours are accessible.

This is a simple contour diagram.

```
In[52]:= ContourPlot[ x y, {x,-1,1}, {y,-1,1}, PlotPoints -> 3,
                    Contours -> 1]
```



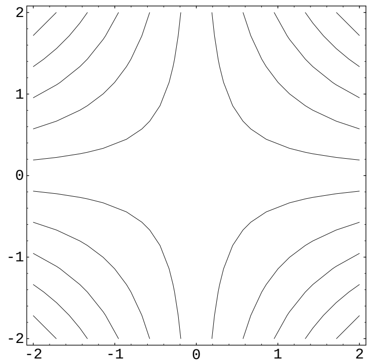
Gaining access to this list is very useful. It can be used for example to construct an equi-potential surface.

```
In[53]:= InputForm[ Graphics[ %]]

Out[53]//InputForm=
  Graphics[{{GrayLevel[0.],
     Polygon[{{-1., 1.}, {1., 1.}, {1., -1.}, {-1., -1.}}]},
    {GrayLevel[1.], Polygon[{{-1., 0.}, {0., -1.},
       {-1., -1.}}], {GrayLevel[0.], Thickness[0.001],
      Line[{{-1., 0.}, {0., -1.}}]}},
    {GrayLevel[1.], Polygon[{{1., 0.}, {0., 1.}, {1., 1.}}],
     {GrayLevel[0.], Thickness[0.001],
      Line[{{1., 0.}, {0., 1.}}]}}},
   {PlotRange -> Automatic, AspectRatio -> 1,
    DisplayFunction -> (Display[$Display, #1] & ),
    ColorOutput -> Automatic, Axes -> False,
    AxesOrigin -> Automatic, PlotLabel -> None,
    AxesLabel -> None, Ticks -> Automatic, Prolog -> {},
    Epilog -> {}, AxesStyle -> Automatic,
    Background -> Automatic, DefaultColor -> Automatic,
    DefaultFont -> {"Courier", 5.5}, RotateLabel -> True,
    Frame -> True, FrameStyle -> Automatic,
    FrameTicks -> Automatic, FrameLabel -> None,
    PlotRegion -> Automatic}]
```

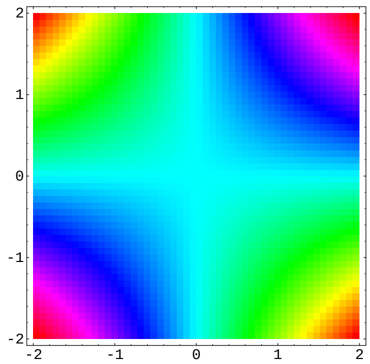
Since `ContourGrahics` objects convert to `Graphics` it is easy to combine them with other plots.

```
In[54]:= ContourPlot[ x y, {x,-2,2}, {y,-2,2},
                         ContourShading -> False]
```



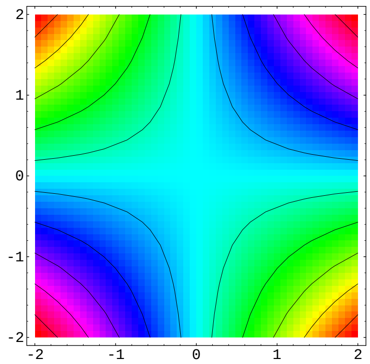
This is a different way to view the same function.

```
In[55]:= DensityPlot[ x y, {x,-2,2}, {y,-2,2},
                    ColorFunction -> Hue,
                    Mesh -> False, PlotPoints -> 50]
```



`Show` makes all the necessary conversions to `Graphics` and combines the resulting primitives.

```
In[56]:= Show[ %, %%]
```



With this ability to combine objects one can build up complex pictures by making use of built-in commands to build individual components. It is always good to use the tools provided by a computer system as much as possible. Thus if, for example, you wish to make some complex `Graphics3D` object try to make parts of it with `Plot3D`.