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

650.12.10.1992

Mathematica Graphics

Tom Wickham-Jones

1 Primitives and Directives

This document is intended to complement the conference notes "Introduction to *Mathematica* Graphics". If the reader has any problems understanding any of the ideas here he is strongly urged to study these introductory notes.

These notes, "*Mathematica* Graphics", attempt to cover two areas. First they look at the actual graphics capabilities themselves, covering topics such as how to use color, how to combine images and the coordinate systems which are used. Secondly they attempt to show how to combine and integrate graphics with the rest of *Mathematica*.

First I shall look at the primitives which can be used to build graphics objects.

■ 1.1 Two-Dimensions

Primitives

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

Two-dimensional graphics primitives.

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}}]]]
```

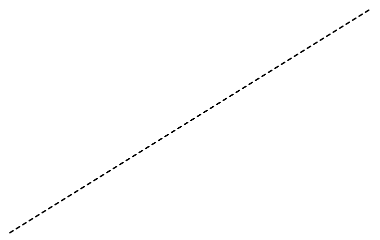
Directives

AbsoluteDashing	AbsolutePointSize
AbsoluteThickness	CMYKColor
Dashing	GrayLevel
Hue	RGBColor
PointSize	Thickness

Two-dimensional graphics directives.

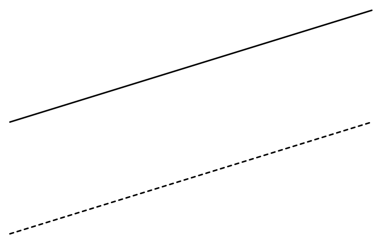
The directive `Dashing` changes the rendering of the `Line` primitive which follows it.

```
In[4]:= Show[ Graphics[ { Dashing[{0.01}],Line[ {{0,0},{1,1}}]}]]
```



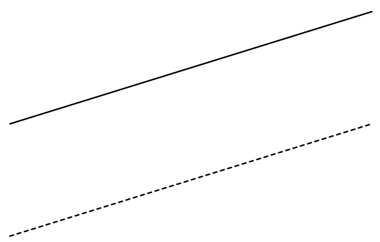
The directive only affects primitives which follow it.

```
In[5]:= 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[6]:= 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.

1.2 Three-Dimensional Primitives and Directives

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

`Graphics3D` directives.

`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. There are some directives which are not present for `Graphics`.

FaceForm and EdgeForm

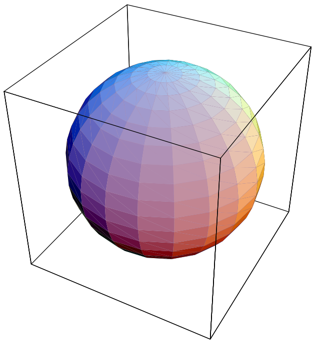
`FaceForm` and `EdgeForm` alter just the edge or the face of a polygon. In addition, `FaceForm` allows the front and back of the polygon to be described differently.

`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 *dirf*

```
In[7]:= prims = First[
         ParametricPlot3D[
         {Sin[t] Cos[p] ,
          Sin[t] Sin[p], Cos[t]},
         {t,0,Pi}, {p, 0, 2Pi},
         DisplayFunction -> Identity]];
```

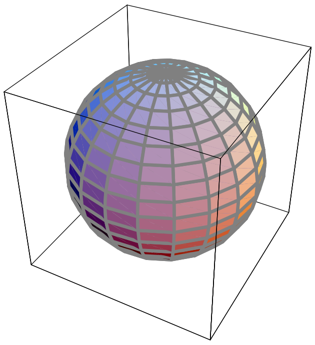
This does not draw lines at the polygon edges.

```
In[8]:= Show[ Graphics3D[ { EdgeForm[ ], prims}]]
```



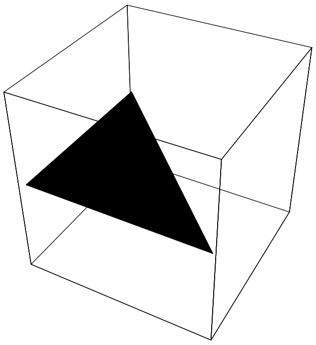
This makes them thick and gray.

```
In[9]:= Show[ Graphics3D[ {
          EdgeForm[ {GrayLevel[0.5], Thickness[0.01]}],
          prims}]]
```



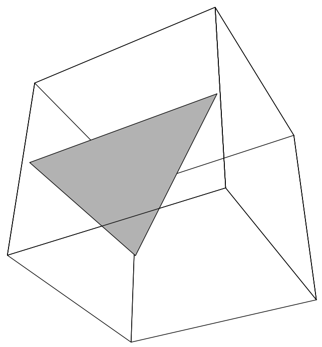
This shows the top view of the polygon.

```
In[10]:= 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[11]:= 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*

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
- ColorOutput

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

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.

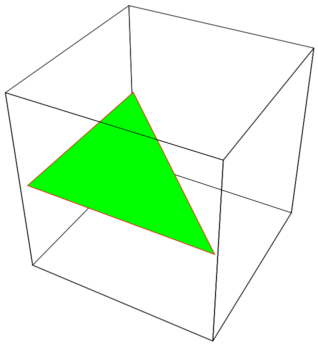
The directives `GrayLevel`, `RGBColor` and `CMYKColor` take as arguments numbers between 0 and 1 inclusive. Thus `GrayLevel[0]` is black, `RGBColor[ 1,0,0]` is a red and `CMYKColor[0,1,0,0]` is a magenta.

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.

```
In[12]:= 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[13]:= Show[ Graphics3D[ {
              EdgeForm[RGBColor[1,0,0]],
              FaceForm[RGBColor[0,1,0]],
              Polygon[{{0,0,0}, {1,0,0}, {0,1,0}}]},
         Lighting -> False]]
```



■ 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.

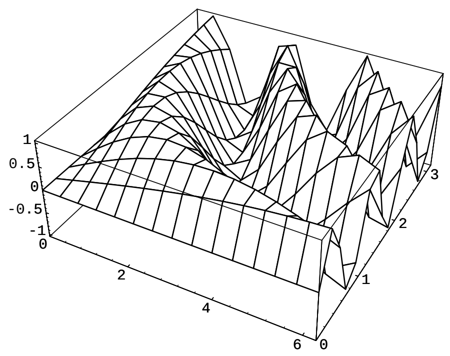
```
In[14]:= Options[ Graphics3D, LightSources]

Out[14]= {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[15]:= Plot3D[ Sin[x y], {x,0,2Pi}, {y,0,Pi}, Shading -> False]
```



■ 2.3 Function Evaluation

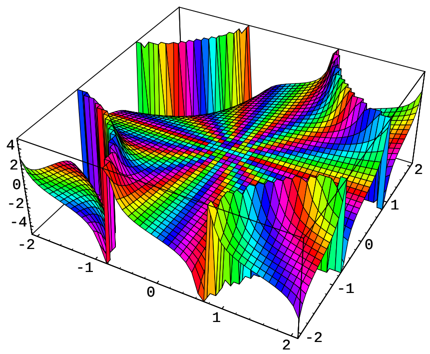
Mathematica provides a number of ways to color a function according to an evaluation.

- `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.

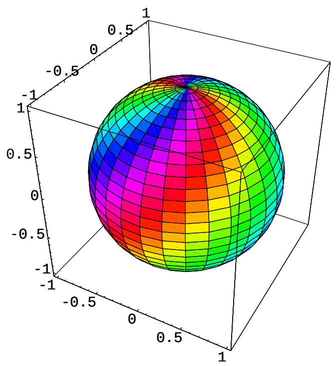
Height is determined by the first element of the list and the color by the second.

```
In[16]:= 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[17]:= 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]
```

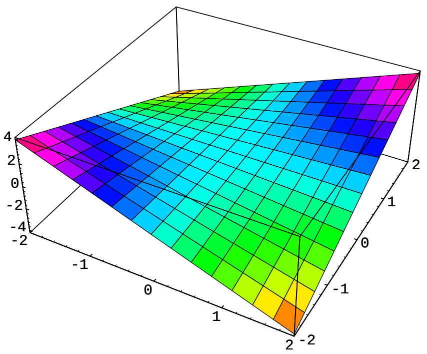


ColorFunction

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

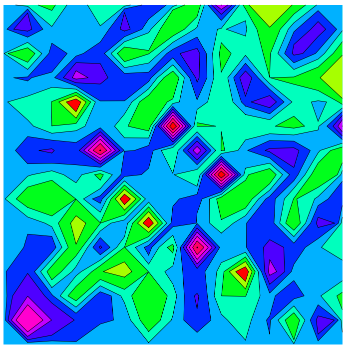


This defines a `ColorFunction`. Note that it is a function which is taking one argument.

```
In[19]:= col[ z_] := Hue[ z^0.65]
```

This looks interesting on a color display.

```
In[20]:= ContourPlot[ Evaluate[
                  Sum[ Sin[n x m y], {n,5}, {m,5}]],
              {x,0,2Pi},{y,0,2Pi}, ColorFunction -> col,
              Frame -> None]
```



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

```
In[22]:= colsep[ RGBColor[ r_, g_, b_]] :=
               CMYKColor[ 1-r, 1-g, 1-b, 0]
```

`colsep` is a simple color separation algorithm.

```
In[23]:= colsep[ GrayLevel[ g_]] :=
               CMYKColor[ 0, 0, 0, g]
```

```
In[24]:= colsep[ RGBColor[ 1,1,0]]

Out[24]= CMYKColor[0, 0, 1, 0]

In[25]:= colsep[ GrayLevel[ 0]]

Out[25]= CMYKColor[0, 0, 0, 0]
```

This uses `colsep`.

```
In[26]:= Show[ plot, ColorOutput -> colsep]
```

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

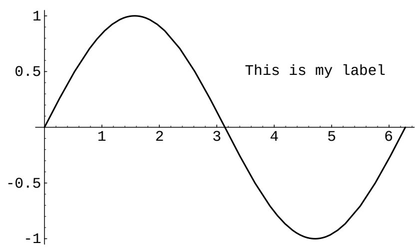
Ways to combine and convert *Mathematica* graphical images.

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

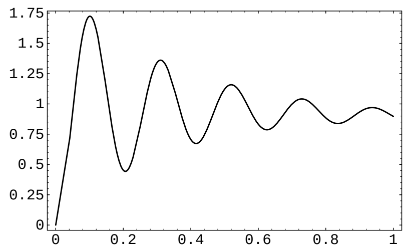


This defines a function `funct` which has a different long time behavior compared with the short time. The last section describes *Mathematica* functions in more detail.

```
In[28]:= funct[ x_] := Exp[ -x/10] - Cos[ 30 x] Exp[ -3x]
```

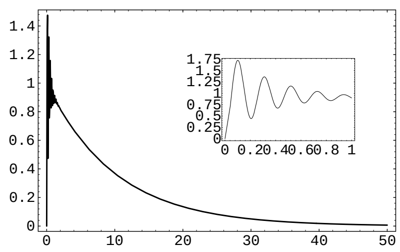
This makes a plot of the short time behavior and stores it as the symbol `short`.

```
In[29]:= 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[30]:= Plot[ funct[x], {x,0,50},
            Frame -> True,
            Prolog -> Rectangle[ {18,.5}, {48,1.2}, short]]
```

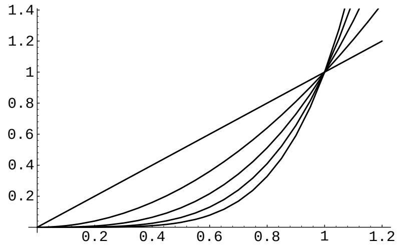


Plotting Several Functions

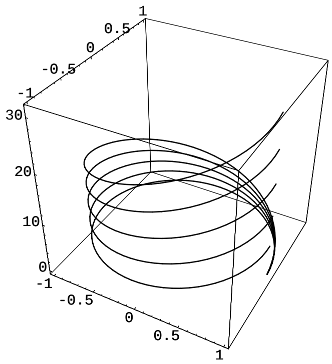
The function plotting commands `Plot`, `ParametricPlot`, and `ParametricPlot3D` all allow more than one function to be plotted.

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

In[31]:= `Plot[ Evaluate[ Table[ x^n, {n,5}]], {x,0,1.2}]`



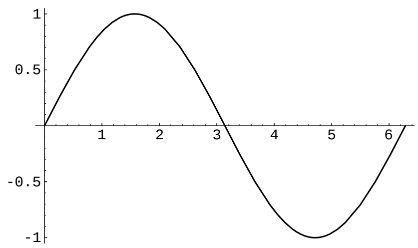
In[32]:=
```
ParametricPlot3D[ Evaluate[
        Table[{ Cos[x], Sin[x], n x},
                {n,5}]],
        {x,0,2Pi}, BoxRatios -> {1,1,1}]
```



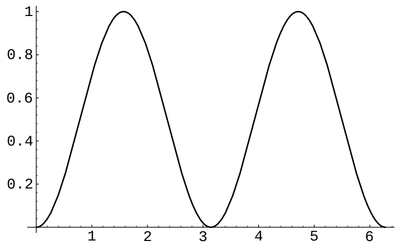
Show

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

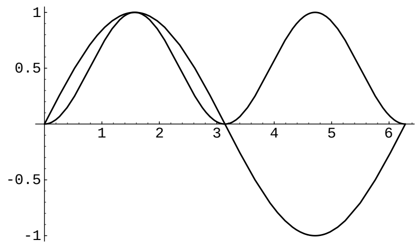
In[33]:= `Plot[ Sin[x], {x,0,2Pi}]`



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



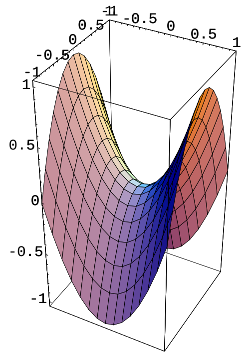
```
In[35]:= Show[%, %%]
```



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`.

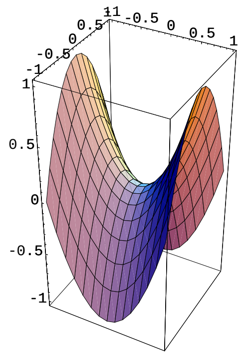
This makes a `SurfaceGraphics` object.

```
In[36]:= plot = Plot3D[ x^2 - y^2 , {x,-1,1}, {y,-1,1},
            BoxRatios -> {1,1,2}]
```



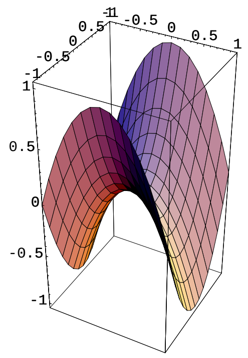
This converts it to `Graphics3D`.

```
In[37]:= Show[ Graphics3D[ %]]
```



This makes another `SurfaceGraphics` object.

```
In[38]:= plot1 = Plot3D[ y^2 - x^2,  {x,-1,1}, {y,-1,1},
                   BoxRatios -> {1,1,2}]
```

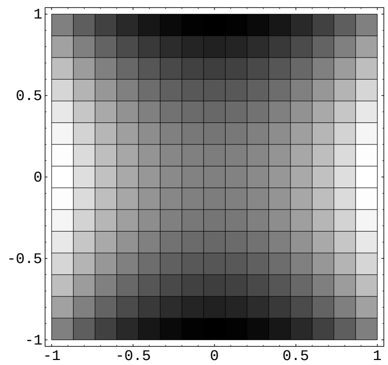


The conversion to `Graphics3D` happens automatically.

```
In[4]:= Show[plot1, plot]
```

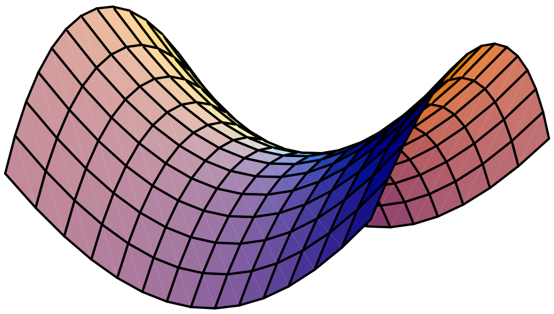
One can convert to other graphics forms as well.

```
In[39]:= Show[ DensityGraphics[ plot]]
```



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

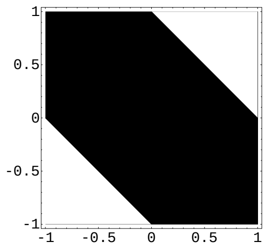
```
In[40]:= Show[ Graphics[ plot]]
```



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[41]:= 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 equipotential surface.

```
In[42]:= InputForm[ Graphics[ %]]

Out[42]//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}]
```

4 Coordinate Specification

`{x, y}`	Two-dimensional coordinates
`Scaled[{x, y}]`	
`Scaled[{dx, dy}, {x, y}]`	
`{x, y, z}`	Three-dimensional coordinates
`Scaled[{x, y, z}]`	
`Scaled[{dx, dy, dz}, {x, y, z}]`	

Types of coordinates for two and three dimensions.

The coordinates for a primitive such as `Line` or `Polygon` can be given in a number of ways. The simplest and most obvious is to give pairs of numbers for two-dimensional graphics and triples for three-dimensional. These are called user coordinates, they are the initial points which we wish to represent. An alternative way to represent points, in a particular part of the picture, is to use `Scaled` coordinates. These refer to display coordinates. For example, they allow us to draw something in the middle of the image without having to know anything about the user coordinates being displayed.

4.1 PlotRange

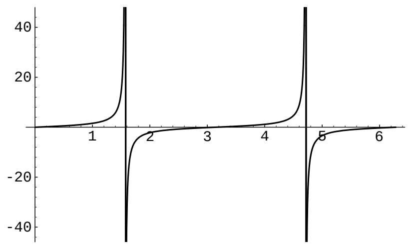
As an object is rendered the list of primitives and directives is scanned to determine the extent of the coordinates which are shown. This is controlled by the option `PlotRange` which provides considerable control of the range which will be seen.

`PlotRange -> Automatic`	only the most interesting part
`PlotRange -> All`	the full range of coordinates
`PlotRange -> {min, max}`	the range `{min -> max}`

Settings for the `PlotRange option`.

This shows a plot of `Tan[x]` with the default value `PlotRange -> Automatic`

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



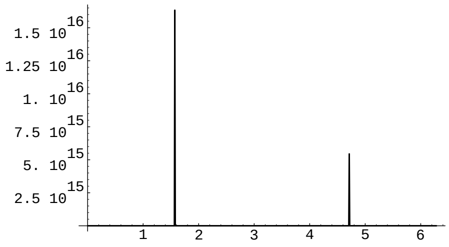
This shows the actual numbers which were used.

```
In[44]:= FullOptions[ %, PlotRange]

Out[44]= {{-0.15708, 6.44026}, {-45.9115, 47.9542}}
```

This plots all the primitives. Notice that this shows less about the `Tan` function than the previous plot.

```
In[45]:= Show[ plot, PlotRange -> All]
```



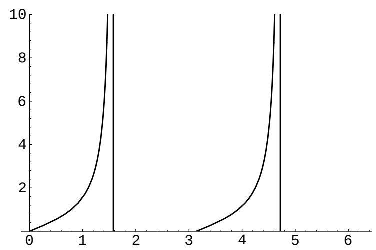
`FullOptions` is a very useful way to find out what `PlotRange` was actually used.

```
In[46]:= FullOptions[ %, PlotRange]

                                          14                 16
Out[46]= {{-0.15708, 6.44026}, {-4.08294 10  , 1.67401 10  }}
```

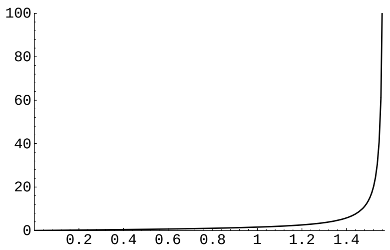
This applies to the y direction only. The x `PlotRange` is determined by the default.

```
In[47]:= Show[ plot, PlotRange -> {0,10}]
```



This works on both dimensions.

```
In[48]:= Show[ plot, PlotRange -> {{0, Pi/2}, {0,100}}]
```



In general, a wide range of settings and combinations for the `PlotRange` option can be used.

`PlotRange -> {Automatic, All}`	two dimensions
`PlotRange -> {All, {5., 10.}}`	two dimensions
`PlotRange -> {Automatic, {3.,5.}, All}`	three-dimensions

Example settings for `PlotRange`.

■ 4.2 Scaled

The examples of primitives which we have seen all showed the coordinates being given in one system, the original or user coordinate system. Sometimes you wish to choose points described in the final or display coordinate system. This is done with the `Scaled` command. Here I describe the two-dimensional form. There is an equivalent three-dimensional version.

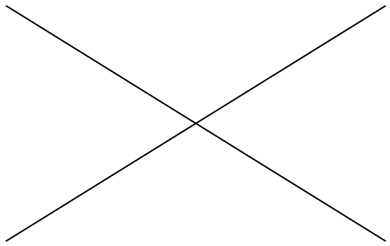
{*x*, *y*}	{*x*, *y*} in user coordinates
`Scaled[{`*sx*, *sy*`}]`	{*sx*, *sy*} in display coordinates
`Scaled[{`*dx*, *dy*`}, {`*x*,*y*`}]`	{*dx*, *dy*} in display coordinates relative to {*x*, *y*} in user coordinates

Coordinates which can be used to build two-dimensional primitives.

`Scaled` coordinates are chosen to run from 0 to 1 in both dimensions and to have their origin at the lower-left corner of the picture. Thus `Scaled[{0,0}]` refers to the minimum of the `PlotRange` and `Scaled[{1,1}]` to the maximum.

This always draws lines from corner to corner.

```
In[49]:= Show[ Graphics[{
         Line[ {Scaled[ {0,0}], Scaled[{1,1}]}],
         Line[ {Scaled[ {0,1}], Scaled[{1,0}]}]}]]
```



This is a *Mathematica* function which uses the offset version of `Scaled`.

```
In[50]:= errorbar[ {x_, y_}] :=
             {Line[{Scaled[ {-.01, -.02}, {x,y}],
                    Scaled[ {.01, -.02}, {x,y}]}],
              Line[{Scaled[ {0, -.02}, {x,y}],
                    Scaled[ {0, .02}, {x,y}]}],
              Line[{Scaled[ {-.01, .02}, {x,y}],
                    Scaled[ {.01, .02}, {x,y}]}]}
```

```
In[51]:= Show[ Graphics[ {
         errorbar[ {.1, .1}], errorbar[ {.9, .9}]}]]
```

■ 4.3 AspectRatio

The aspect ratio of the graphics object is set by the option `AspectRatio`.

`AspectRatio -> ` *num*	show picture with a height/width = *num*
`AspectRatio -> Automatic`	show picture with ratio taken from data

Settings for the `AspectRatio` option.

This uses the default `AspectRatio` of `1/GoldenRatio`, which is equal to 0.618034.

In[52]:= `Show[ Graphics[ Rectangle[ {0,0}, {4,1}]]]`

This will produce a square image.

In[53]:= `Show[ %, AspectRatio -> 1]`

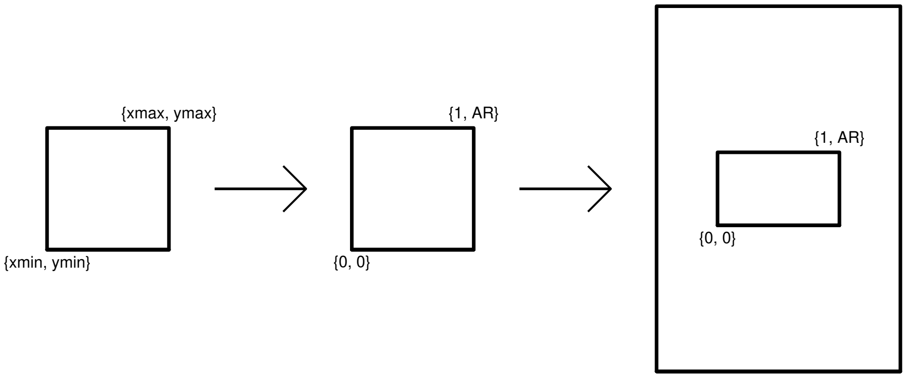
This shows the real x to y proportions.

In[54]:= `Show[ %, AspectRatio -> Automatic]`

Even though `AspectRatio -> Automatic` maintains the x to y scaling, this is often not very useful. For example `Plot[ Tan[x], {x,0,2Pi}, AspectRatio -> Automatic]`.

■ 4.4 The Coordinate Transformations

Once the range of coordinates, $\{xmin, ymin\}$ to $\{xmax, ymax\}$ that will be shown is determined, the actual mapping onto the screen or paper has to be carried out. There are two transformations in the process by which a *Mathematica* graphics object is converted into a sequence of painting instructions on a screen or paper. The first involves a transformation of the user coordinates, set by `PlotRange`, into display coordinates such that $xmin, ymin$ and $xmax, ymax$ map to $\{0, 0\}$ and $\{1, AspectRatio\}$ respectively. Once that is done the display coordinates are mapped onto the paper so as to leave room for axes, labels and margins.



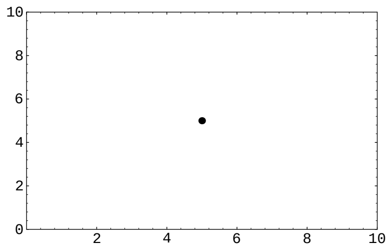
Coordinate system transformations.

This second transformation is quite complex and often leads to confusion. It makes the coordinate area, into which user coordinates are drawn, to be rectangular, to have a given aspect ratio and to be as large as possible. This is done by emitting all of the axes and labels, for which space must be left, before any other objects. This initial part of the stream is called the critical region. All objects which are emitted in the critical region will be included in the calculations which map the coordinate area onto the window. Text which is associated with axes, such as tick and axes labels, will be included in the critical region.

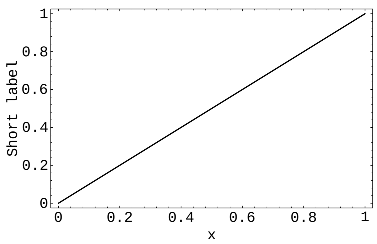
Eventually the critical region is closed and objects which are emitted will no longer be used for scaling. This is usually after the axes have been drawn. The objects which follow will be clipped against the coordinate area.

The point is mapped into the middle of the display coordinate space. Notice the frame axes that mark the bounds of the coordinate area. These are emitted in the critical region.

```
In[55]:= Show[ Graphics[ {
              PointSize[0.02],
              Point[{5,5}]},
              PlotRange -> {{0,10}, {0,10}},
              Frame -> True]]
```

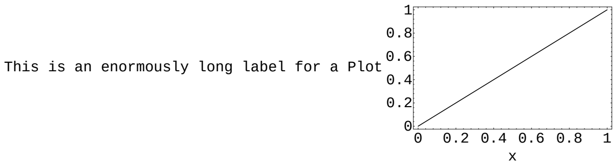



```
In[56]:= Plot[ x, {x,0,1}, Frame -> True,
           FrameLabel -> {"x", "Short label"}]
```



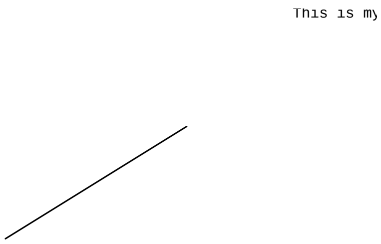
Notice that the plot has shrunk to include the label.

```
In[57]:= Show[%,
        FrameLabel -> {"x",
        "This is an enormously long label for a Plot"},
        RotateLabel -> False]
```



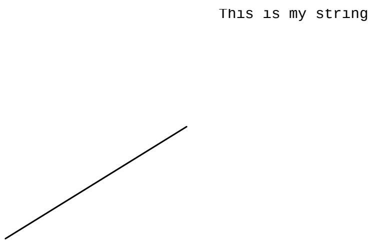
The text is cut at the point {2,2} since this is the edge of the coordinate area.

```
In[58]:= Show[ Graphics[{
      Line[{{0,0}, {1,1}}],
      Text["This is my string", {2,2}]}]]
```



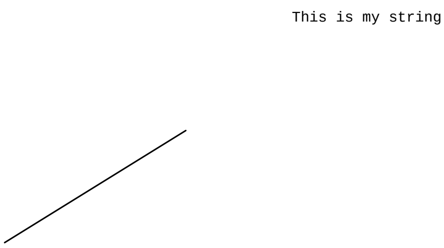
The text is complete since the offset of the `Text` primitive has moved it into the coordinate area.

```
In[59]:=
    Show[ Graphics[{
            Line[{{0,0}, {1,1}}],
            Text["This is my string", {2,2}, {1,0}]}]]
```



The string is allowed to spill out of the coordinate area since the text was emitted in the critical region.

```
In[60]:=
    Show[ Graphics[{
            Line[{{0,0}, {1,1}}],
            Text["This is my string", {2,2}]},
            PlotRange -> All]]
```

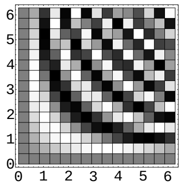


■ 4.5 PlotRegion

The option `PlotRegion` allows one to add margins around a plot.

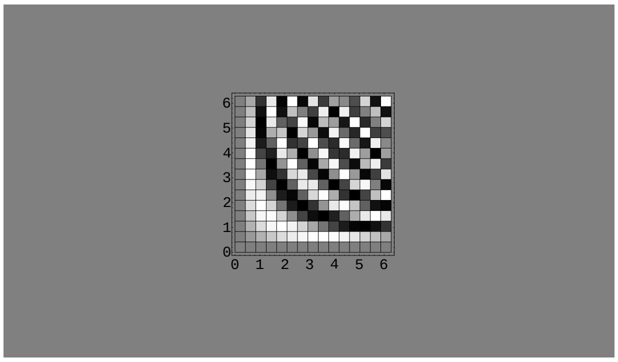
The coordinates for `PlotRegion` are `Scaled` coordinates which run from 0 to 1 in each dimension.

```
In[61]:= DensityPlot[ Sin[x y],
                      {x, 0,2Pi},
                      {y,0,2Pi},
                      PlotRegion -> {{0.25,0.75},
                                     {0.25,0.75}}]
```



Showing the background shows how the image has been shrunk.

```
In[62]:= Show[%, Background -> GrayLevel[0.5]]
```



5 Three-Dimensional Coordinate Systems

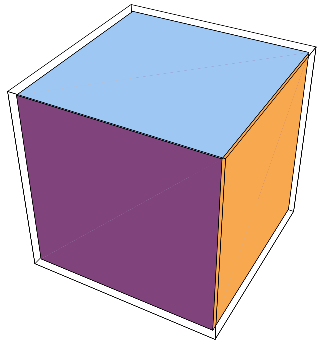
Now we shall look at the parts of the coordinate system transformation which are particular to three-dimensional objects.

■ 5.1 BoxRatios

The `PlotRange` `{{xmin, xmax}, {ymin, ymax}, {zmin, zmax}}` is determined as the first stage of processing a list of primitives. This determines the box into which the primitives will be drawn. This also is the box drawn with the `Boxed` option. The ratios of the lengths of the sides of this box are controlled with the option `BoxRatios`. This is the three-dimensional analog of the `AspectRatio`.

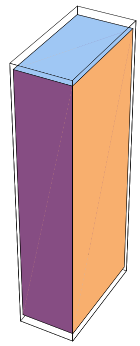
This shows a cubical image since the ratios of the box are all the same.

```
In[63]:= Show[ Graphics3D[
              Cuboid[{0,0,0}, {1,2,5}],
              BoxRatios -> {1,1,1}]]
```



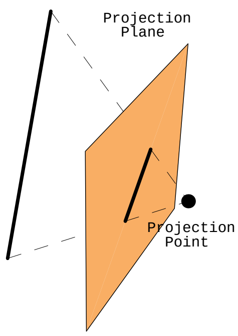
This shows the actual ratios of the `Cuboid` dimensions.

```
In[64]:=
      Show[ Graphics3D[ Cuboid[{0,0,0}, {1,2,5}],
              BoxRatios -> Automatic]]
```



5.2 Projection

The process by which a coordinate system of dimension n is transformed into one of dimension less than n is called a projection. In order to view a three-dimensional object on a computer screen or a printed page it is necessary to project it from three to two dimensions.

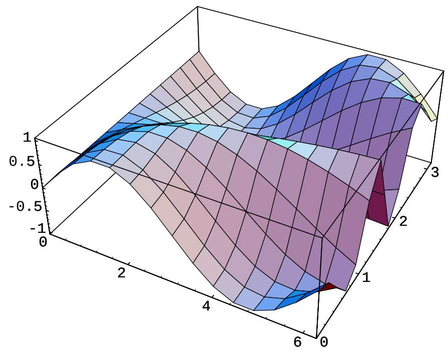


Projection of a line

The components which define a projection are a projection plane, which defines the two-dimensional space to which we wish to move, and a projection point. The projection is made by forming the straight line from the projection point to the object we wish to project. Where this line cuts the projection plane is the projection of that object.

In *Mathematica* the projection point is given by the option `ViewPoint` and the projection plane is defined by the plane which contains the `ViewCenter` and is perpendicular to the vector from the `ViewPoint` to the `ViewCenter`, the line-of-sight vector.

```
In[65]:= Plot3D[ Sin[ x  Cos[y]], {x,0,2Pi}, {y,0,Pi}]
```



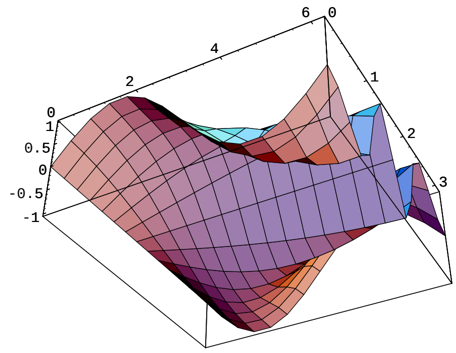
This is the default `ViewPoint` which was used.

```
In[66]:= FullOptions[%, ViewPoint]

Out[66]= {1.3, -2.4, 2.}
```

This reflects the viewpoint in the z-direction to show us the underside of the surface.

```
In[67]:= Show[ %%, ViewPoint -> % {1,1,-1}]
```

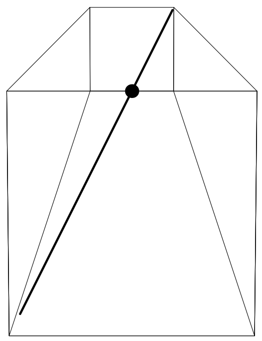


`ViewPoint` is given in a special scaled coordinate system in which the origin is the center of the box. The units are scaled so that the longest side of the box is unity and the other dimensions follow their `BoxRatios`. Note that the `ViewPoint` must be outside of the box. This choice of coordinate system allows us to gain an interesting view of a wide variety of three-dimensional objects from one default `ViewPoint`.

`ViewCenter` is used to set the projection plane as is specified above. It is also used to specify which part of the projected image will appear in the center of the window. The coordinate system of `ViewCenter` is `Scaled` coordinates, the lower-left corner of the box is `{0,0,0}` and the upper-right corner is `{1,1,1}`. With the default setting `ViewCenter -> Automatic` *Mathematica* chooses the projection plane to pass through the center of the box but places the center of the projection so that the image is as large as possible. `ViewCenter -> {0.5, 0.5, 0.5}` uses the same projection plane but puts the point `Scaled[{0.5, 0.5, 0.5}]` at the center of the drawing region.

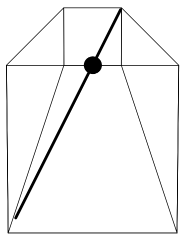
This uses the default `ViewCenter -> Automatic`.

```
In[68]:= Show[ Graphics3D[ {
             PointSize[0.05],
             Point[{.5,5,.5}],
             Thickness[0.008],
             Line[{{0,0,0}, {1,10,1}}]},
             ViewPoint -> {0, -1, .1}]]
```



See how the image has shrunk since the center of the image is at the center of the drawing region here.

```
In[69]:= Show[ %, ViewCenter -> {0.5, 0.5, 0.5}]
```



SphericalRegion

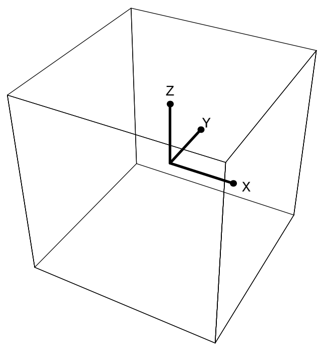
With the default setting of `SphericalRegion -> False` the image is scaled so that the projection of the box fits into the window. Setting `SphericalRegion -> True` scales so that the sphere which can be transcribed around the box fits the window. This is desirable if one wishes to animate the rotation of a graphic by changing the `ViewPoint`.

ViewVertical

The option `ViewVertical` is provided to allow the user to specify which direction in the image will point up in the projected image. The default value of this is `ViewVertical -> {0,0,1}` so the z-direction (at the `ViewCenter`) points up.

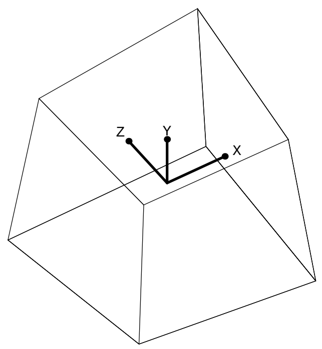
Z points up here.

```
In[70]:= Show[
            Graphics3D[ {PointSize[.02],
              Thickness[.008], Line[ {{0,0,0}, {1,0,0}}],
              Point[{1,0,0}], Text[ "X", {1.2,0,0}],
              Line[ {{0,0,0}, {0,1,0}}], Point[{0,1,0}],
              Text[ "Y", {0,1.2,0}], Line[ {{0,0,0}, {0,0,1}}],
              Point[{0,0,1}], Text[ "Z", {0,0,1.2}]},
              PlotRange -> {{-1.5,1.5}, {-1.5,1.5},
                                       {-1.5,1.5}}]]
```



This makes Y point up.

```
In[71]:= Show[ %, ViewVertical -> {0,1,0}]
```



AspectRatio

With the specification of `BoxRatios`, `ViewPoint`, `ViewCenter`, `ViewVertical`, and `SphericalRegion` the projection from three-dimensional coordinates is complete. However, the relative scaling of x and y in the projection plane can be altered with the option `AspectRatio`. This defaults to `Automatic`, which maintains the relative proportions which were initially present. Under most circumstances this is desirable.

6 Graphics Programming

One of the most powerful aspects of *Mathematica* graphics is the availability of the rest of the system. This provides many capabilities and functions which are not normally present in a graphics application. One can use the built-in *Mathematica* functions which implement a wide range of numerical techniques such as root-finding, function minimization, numerically solving differential equations or quadrature, and then graph the results. Further still one can write code to construct primitives which draw a special type of picture.

This chapter provides some introduction to the use of graphics with the rest of the system and some idea of graphics programming. First we shall look at some ways to program in *Mathematica*. For more information on *Mathematica* programming consider the book *Mathematica: A System for Doing Mathematics by Computer*, Second Edition, by Stephen Wolfram, *Mathematica* conference notes which are available from Wolfram Research or books on the subject such as *Programming in Mathematica* by Roman Maeder or *Mathematica: A Practical Approach* by Nancy Blachman.

6.1 Introduction to *Mathematica* Programming

Mathematica Patterns and Rules

One of the unique things about *Mathematica* programming is the ease and economy with which rules are given to describe how a function should behave. If you don't like the way it works, add a rule to make it work just as you desire.

f[1]	matches exactly *f*[1]
f[_]	matches *f*[*thing*]
f[x_]	matches *f*[*thing*] and gives *thing* a name

Some simple *Mathematica* patterns.

This gives a rule for `f[1]`.

```
In[72]:= f[1] := foo
```

This evaluates since a rule matches the input.

```
In[73]:= f[1]

Out[73]= foo
```

This does not evaluate since there is no rule to match it.

```
In[74]:= f[2]

Out[74]= f[2]
```

This will match `f[` *thing* `]`.

```
In[75]:= f[ _] := bar
```

Now this matches a pattern and the result is returned.

```
In[76]:= f[2]

Out[76]= bar
```

This does not match since a pattern has only been for `f` of one argument.

```
In[77]:= f[ 2,3]

Out[77]= f[2, 3]
```

This will clear all the rules of `f`.

```
In[78]:= Clear[ f]
```

This gives *thing* a name, namely `x`.

```
In[79]:= f[ x_] := x^3 - x^2
```

This matches.

```
In[80]:= f[ 5]

Out[80]= 100
```

And this matches the pattern as well.

```
In[81]:= f[ (a + b)]

                    2          3
Out[81]= -(a + b)  + (a + b)
```

More complex patterns can be given to match multiple arguments, to refer to particular types of inputs and to accept conditions. The book *Mathematica: A System for Doing Mathematics by Computer*, Second Edition, describes patterns is considerably more detail.

Delayed Evaluation

Often in these notes we see = and := which are the assignment functions. The first evaluates the right-hand side before the assignment is made. The second, := which is delayed assignment, does not evaluate the right-hand side until the left-hand side is evaluated.

=	evaluate and then assign
:=	assign but don't evaluate

This evaluates before assignment.

```
In[82]:= f1 = Random[]

Out[82]= 0.0560708
```

These are all the same since `f1` has been set to the value of the random number.

```
In[83]:= Table[ f1, {5}]

Out[83]= {0.0560708, 0.0560708, 0.0560708, 0.0560708, 0.0560708}
```

This makes a delayed assignment.

```
In[84]:= f2 := Random[]
```

We now see a list of different and random numbers. This is because the evaluation was delayed.

```
In[85]:= Table[ f2, {5}]

Out[85]= {0.6303, 0.359894, 0.871377, 0.858645, 0.584579}

In[86]:= ??f1

Global`f1
f1 = 0.05607079483486153
```

The settings of the ?? clearly show the difference.

```
In[86]:= ??f2

     Global`f2
f2 := Random[]
```

Delayed assignment avoids evaluating the left-hand side. There are a number of other functions which do not evaluate their arguments. An example is `Plot`, which possesses the attribute `HoldAll` which delays the evaluation of arguments.

`HoldAll`	a `HoldAll` function will delay argument evaluation
`Evaluate[` *expr*`]`	evaluate this even if in a `HoldAll` function
`Hold[` *expr* `]`	do not evaluate *expr*
`ReleaseHold[` *expr*`]`	remove `Hold` if present

Other ways to delay evaluation.

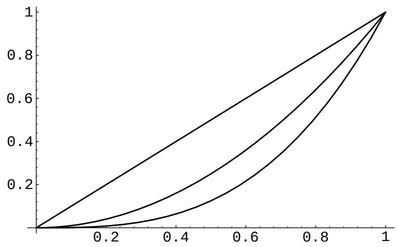
This shows that `Plot` delays evaluation of the arguments.

```
In[86]:= Attributes[ Plot]

Out[86]= {HoldAll, Protected}
```

This would not plot three functions without the `Evaluate`.

```
In[87]:= Plot[ Evaluate[ Table[ x^n, {n,3}]], {x,0,1}]
```



This does not evaluate due to `Hold`.

```
In[88]:= Hold[ 1 + 5]

Out[88]= Hold[1 + 5]
```

`ReleaseHold` removes the `Hold` and evaluation proceeds.

```
In[89]:= ReleaseHold[ %]

Out[89]= 6
```

Working with Lists

Mathematica lists are fundamental to the whole system. A good understanding of the power of list manipulations is a key to an understanding of *Mathematica* itself. There are many functions which work with lists and here we review some of the most basic.

`Table`	construct a list
`Map[` *f*`,` *list*`]`	travel down *list* applying *f* to each element
`Apply[` *f*`,` *list*`]`	make *f* the head of *list*
`Transpose[ {`*list1*`,` *list2*`}]`	transpose `{`*list1*`,` *list2*`}`

Some useful commands for working with lists.

Make a list.

```
In[90]:= list = Table[ i, {i,5}]

Out[90]= {1, 2, 3, 4, 5}
```

This travels down the list applying g to everything. It returns a list as the result.

```
In[91]:= Map[ g, list]

Out[91]= {g[1], g[2], g[3], g[4], g[5]}

In[92]:= Map[ Sqrt, list]

Out[92]= {1, Sqrt[2], Sqrt[3], 2, Sqrt[5]}
```

This exchanges the head of list with g.

```
In[93]:= Apply[ g, list]

Out[93]= g[1, 2, 3, 4, 5]
```

Of course the result may evaluate further. Using `Times` will multiply all the numbers together.

```
In[94]:= Apply[ Times, list]

Out[94]= 120

In[95]:= list1 = {a,b,c}

Out[95]= {a, b, c}

In[96]:= list2 = {1,2,3}

Out[96]= {1, 2, 3}
```

This is an extremely useful way to collect things together.

```
In[97]:= Transpose[ {list1, list2}]

Out[97]= {{a, 1}, {b, 2}, {c, 3}}
```

Blocks and Modules

If you have lines of code with a common purpose and wish to group them together this may be done with either `Block` or `Module`. These allow you to collect commands together and provides tools for local variables.

- `Block[ {`*locals*`}, `*com1*`; ... `*com2*`]`
- `Module[ {`*locals*`}, `*com1*`; ... `*com2*`]`

The result of `Block` is the last evaluation.

```
In[98]:= blockfun[  n_] :=
            Block[ { i, res},
              res = Table[ i, {i, n}] ;
              res = Partition[ res, 2] ;
              Transpose[ res]
            ]

In[99]:= blockfun[ 10]

Out[99]= {{1, 3, 5, 7, 9}, {2, 4, 6, 8, 10}}
```

`Module` is very similar to `Block`. The difference is the treatment of local variables.

```
In[100]:= modfun[  n_] :=
            Module[ { i, res},
              res = Table[ i, {i, n}] ;
              res = Partition[ res, 2] ;
              Transpose[ res]
            ]

In[101]:= modfun[ 12]

Out[101]= {{1, 3, 5, 7, 9, 11}, {2, 4, 6, 8, 10, 12}}
```

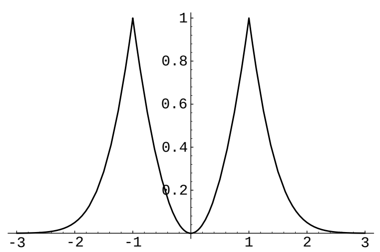
■ 6.2 Function Plotting

Constructing a new function is very simple:

```
In[102]:= weirdf[x_] := If[ x > -1 && x < 1, x^2, Exp[ 1- x^2] ]
```

The function which `Plot` requires must evaluate to a number. Thus our weird function is quite adequate.

```
In[103]:= Plot[ weirdf[t], {t,-3,3}]
```



A further possibility is to use *Mathematica* numerical capabilities. For this the `NDSolve` function, which numerically solves differential equations, is very suitable.

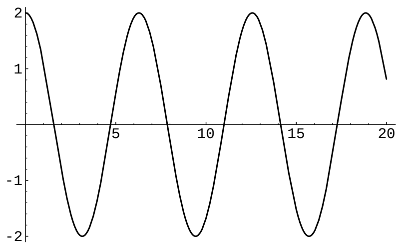
`NDSolve` returns an `InterpolatingFunction`. Of course we could have solved this problem analytically.

```
In[104]:= NDSolve[ {x''[t] == -x[t],
                    x[0] == 2,
                    x'[0] == 0}, x, {t,0,20}]

Out[104]= {{x -> InterpolatingFunction[{0., 20.}, <>]}}
```

This plots the result. `Evaluate` is necessary since `Plot` is `HoldAll`.

```
In[105]:= Plot[ Evaluate[ x[t] /. %], {t,0,20}]
```



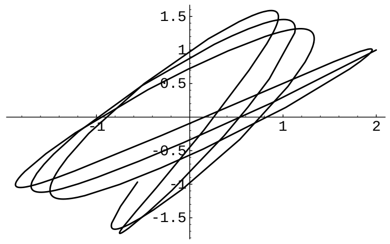
This example is more complex. It comprises of two coupled oscillators.

```
In[106]:= NDSolve[ {x''[t] == -x[t] - x[t] y[t]^2,
                    y''[t] == -y[t] -x[t] y[t]^2,
                    x[0] == 2, x'[0] == 0,
                    y[0] == 1, y'[0] == 0},
                    {x,y}, {t,0,20}]

Out[106]= {{x -> InterpolatingFunction[{0., 20.}, <>],
   y -> InterpolatingFunction[{0., 20.}, <>]}}
```

This shows the solution moving between x and y.

```
In[107]:= ParametricPlot[ Evaluate[ {x[t], y[t]} /. First[%]],
          {t,0,20}]
```



■ 6.3 Animation

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

In *Mathematica* there are a number of ways to carry out animations. The simplest method is available for the Notebook front end to *Mathematica*. One generates a sequence of graphics, selects them and issues the appropriate command to start the animation. The other way, also available, on front ends which do not support Notebooks is to use the package `` Graphics`Animation` ``. This package defines useful commands which work on Notebook and non-Notebook front ends.

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

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

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

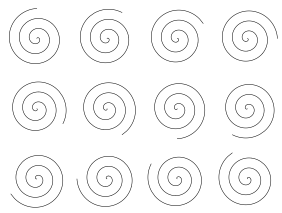
Do not load `FlipBookAnimation`` unless you wish to make a static sequence of pictures like these. If you wish an animated image your system is already configured to work.

```
In[108]:= <<FlipBookAnimation`
```

```
In[109]:= <<Graphics`Animation`
```

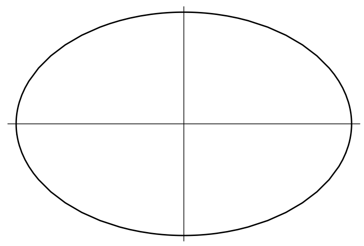
Here both `PlotRange` and `AspectRatio` are set. This is to enhance the effect of the animation.

```
In[110]:= Animate[
            ParametricPlot[ {t Sin[t+n], t Cos[t+n]}, {t,0,6Pi},
              AspectRatio -> Automatic,
              PlotRange -> {{-20, 20},{-20,20}}, Axes -> False],
            {n,0,2Pi-Pi/6, Pi/6}]
```



As shown here it is very often necessary to give explicit settings to `PlotRange`. This ensures that objects with the same coordinates do not move from frame to frame. As another example of an animation let's consider an ellipse:

```
In[111]:= plot = ParametricPlot[ { 1.5 Sin[x] , Cos[x]},
          {x,0,2Pi},
                AspectRatio -> Automatic, Ticks -> None]
```



This makes a list of primitives which will move a point around the edge of the ellipse. It will also join lines from the foci to this point. In a real application it may be interesting to decrease the step-size.

```
In[112]:= prims =
        Table[{ PointSize[ 0.04],
                Point[{1.5 Sin[t], Cos[t]}],
                Thickness[ 0.01],
         Line[ {{-Sqrt[ 1.5^2 - 1.], 0},
                       {1.5 Sin[t], Cos[t]},
                       {Sqrt[ 1.5^2 - 1.], 0}}]},
         {t,0,2Pi-Pi/4,Pi/4}];
```

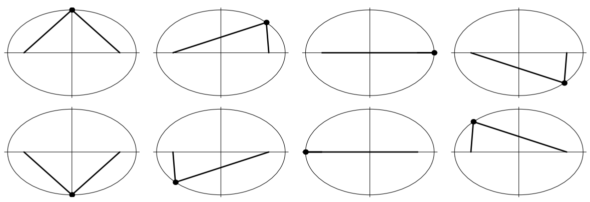
This produces a list of graphics objects, each of which is a different frame.

```
In[113]:= tab = Map[ (plot /. (Epilog -> {}) -> (Epilog -> #))&,
          prims]

Out[113]= {-Graphics-, -Graphics-, -Graphics-, -Graphics-,
  -Graphics-, -Graphics-, -Graphics-, -Graphics-}
```

Now we can show the result with the command `ShowAnimation`.

```
In[114]:= ShowAnimation[ tab]
```



For animations of the `ViewPoint` of three-dimensional images one may wish to use the option `SphericalRegion -> True`. This maintains the same scaling as the object is rotated.

■ 6.4 Examples of Graphics Programming

Having seen how to construct functions for built-in commands we can go further and build our own graphics commands. Commands such as `Plot` evaluate their input function over a range and construct a collection of graphics primitives. We could do something very similar. Of course it would not be so useful to reproduce the actual built-in commands. However there are many other tasks to be tackled by writting code in the *Mathematica* programming language. Here we shall look at some possibilities.

Iterative Maps

The first idea comes from iteration. If one repeatedly applies a function, the values returned in some cases converge to a fixed point and in others to seemingly random and complex behavior. For example if we have a function such as $4x(1-x)$

Floating-point numbers are used to ensure real arithmetic.

```
In[115]:= fun[x_] := 4. x ( 1. -x )
```

This is the value for an argument of .25

```
In[116]:= fun[ .25]

Out[116]= 0.75
```

This gives me 4 iterations of f.

```
In[117]:= fun[ fun[ fun[ fun[ .25]]]]

Out[117]= 0.75
```

Of course *Mathematica* gives a much neater way to apply this function repeatedly.

```
In[118]:= Nest[ fun, .25, 10]

Out[118]= 0.75
```

`NestList` keeps the intermediate values.

```
In[119]:= NestList[ fun, .25, 10]

Out[119]= {0.25, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75, 0.75,
  0.75, 0.75, 0.75}
```

Now we can change the starting point and we see the function does not converge anymore.

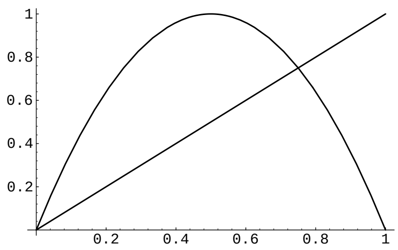
```
In[120]:= NestList[ fun, .24, 10]

Out[120]= {0.24, 0.7296, 0.789135, 0.665603, 0.890303,
  0.390655, 0.952175, 0.182151, 0.595887, 0.963222, 0.1417}
```

Very nice, but I want to see a graphical representation of this. To do this I shall write some code. The function is being applied to some point, and the value which results is used as the next starting point. We are thus interacting with the line $< x, x >$ and we are interested in the two lines $< x, x >$, and $< x, f[x] >$.

These are the two lines we are interested in.

```
In[121]:= plot = Plot[ {x, fun[x]}, {x,0,1}]
```



This function collects all the steps which we have carried out. It takes two arguments and returns a `Graphics` object.

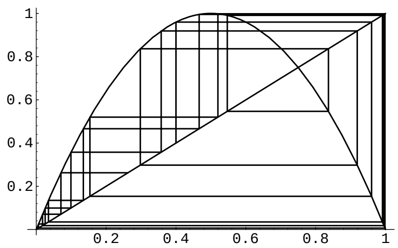
```
In[122]:= Iterate[ st_, n_] :=
          Block[{ res},
            res = NestList[ fun, st, n] ;
            res = Flatten[ Transpose[ {res, res}]] ;
            res = Partition[ res, 2,1] ;
            Graphics[ Line[ res]]
          ]

In[123]:= Iterate[ .4, 20]

Out[123]= -Graphics-
```

This combines the result.

```
In[124]:= Show[ plot, %]
```



In a few lines of code we have a useful way to explore these iterated procedures. This function `Iterate` could be extended to take the iterated function as an argument and also to generate all the plots automatically.

Correlation Plots

As an another example I shall try to build a new type of graphic. *Mathematica* comes with many packages to generate particular types of graphics such as piecharts, barcharts or logplots. We shall look here at another type of plot: one which looks for correlation between sets of data by graphical means.

This is a simple data set.

```
In[125]:= data = Table[ {x, x^2}, {x,5}]

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

This splits up the list.

```
In[126]:= res =
            Table[ Map[ {Part[ #, i], Part[#, j]}&, data],
              {i, 2}, {j, 2}] ;
```

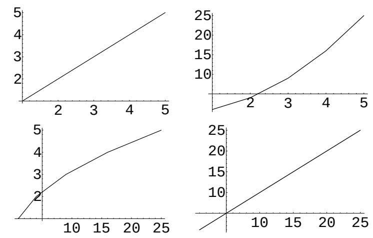
This makes a `GraphicsArray` of the result.

```
In[127]:= res = GraphicsArray[
              Map[ ListPlot[#,
                        PlotJoined -> True,
                        DisplayFunction -> Identity]&,
              res, {2}]]

Out[127]= -GraphicsArray-
```

We can show the result.

```
In[128]:= Show[ res]
```



This is a more complex version. It reverses the data so that the diagonal goes from lower-left to upper-right. It also sets graphics options in the result.

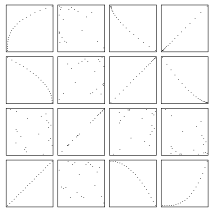
```
In[129]:= CorrelationPlot[ data_] :=
            Block[{ i, j, res, len = Length[ First[ data]]},
              res = Table[ Map[ {Part[ #, i], Part[#, j]}&,
          data],
                  {i, len}, {j, len}] ;
              res = Reverse[ res] ;
              res = Map[ ListPlot[ #, Frame -> True,
                  FrameTicks -> None,
                  AspectRatio -> 1,
                  DisplayFunction -> Identity]&,
                  res, {2}] ;
              Show[ GraphicsArray[ res]]
            ]
```

This is a more complex data set. We shall suppress the output.

```
In[130]:= data = Table[ {x, Random[], -x^2, x^3}, {x,0,1,.05}];
```

This is the result. It neatly shows the relationships between the sets of data.

```
In[131]:= CorrelationPlot[ data]
```



A *MathLink* **Renderer**

As a final suggestion for programming *Mathematica* graphics one may wish to consider using *MathLink* to write your own renderer. *MathLink* will send the `FullForm` of an arbitary *Mathematica* expression to an external process. For example, you could make a function which will render a `Graphics` object and take advantage of some specialisation which *Mathematica* does not normally support. Then you could make your `$DisplayFunction` call this function and all the other commands such as `Plot` and `Show` will just work as before.