8. Graphics Programming

8.1 Introduction

What is graphics programming?

In the Mathematica programming language, the complete information required for the software to render a graph, or graphics object is an *expression*, just like everything else. Thus, we can manipulate graphics objects just like we have been manipulating other expressions.

Some of the things you can do are
- Combine plots together.
- Create your own combination of graphics objects (lines, circles, text, …).
- Add titles, grids, new scales, thicker lines, … to plots.
- Add colour.
- Work in 2 or 3 dimensions.
- Perform image analysis.

Most often you will be designing a function which takes some input, and generates a plot of some sort

Just as the command **`Plot`** produces a two-dimensional **`Graphics`** object, all the graphics commands produce different *graphics objects*. The types of graphics object are

- **`Graphics`** For 2D graphics
- **`Graphics3D`** For 3D graphics
- **`SurfaceGraphics`** For graphics showing surfaces of 3D objects
- **`ContourGraphics`** For showing contour plots of 3D surfaces
- **`DensityGraphics`** For plotting a variable as print density rather than height
- **`GraphicsArray`** For plotting arrays of graphs

We begin by exploring 2D graphics.

Objectives

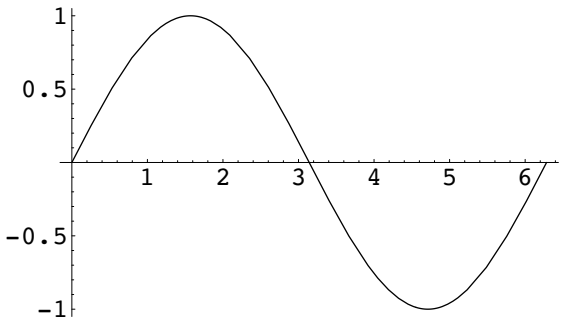
- To understand how graphics objects are encoded in Mathematica
- To be able to write functions which produce a graphic output without unwanted graphic side-effects
- To be able to use graphics options and style directives
- To be familiar with Mathematica's animation capabilities
- To be familiar with Mathematica's graphics packages

8.2 Showing Graphics Objects

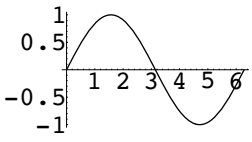
The plot as a side-effect

We start with a simple plot.

```
p = Plot[Sin[x],{x,0,2π}]
```




```
- Graphics -
```



The first thing to understand is that the actual plot that we see, that is , is only produced by the Kernel for our benefit to see, and is no more than a *side-effect* of the output of the graphics object expression. The graphics object expression is given by the output **-**Graphics**-** under the plot, which we have assigned to the value **p**.

If we enter **p**, we simply get back **-**Graphics**-**.

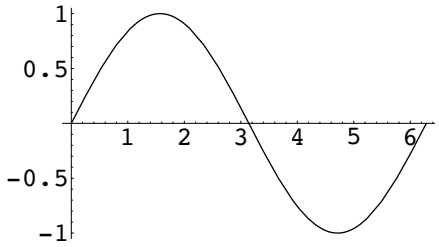
```
p
```

```
- Graphics -
```


Showing plots

If you want to *see* the plot represented by the expression **p**, you have to ask Mathematica to **Show** it to you.

```
Show[p]
```

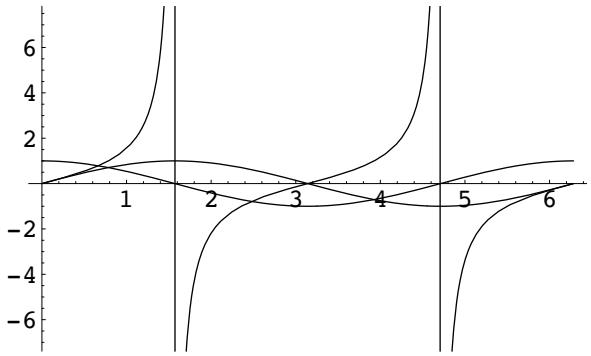



```
- Graphics -
```

If you have several plots p_1, p_2, p_3, …, you can show them all overlaid on top of each other with **Show[p_1, p_2, p_3, …]**. For example

```
p1 = Plot[Sin[x], {x, 0, 2 π}];
p2 = Plot[Cos[x], {x, 0, 2 π}];
p3 = Plot[Tan[x], {x, 0, 2 π}];

Show[p1, p2, p3]
```




```
- Graphics -
```

Show plots p_1, p_2, p_3, …, with **Show[p_1, p_2, p_3, …]** or **Show[{p_1, p_2, p_3, …}]**

- Plots can be resized by clicking on them to select them, and then dragging one of the handles you see.
- You can also copy and paste them into text.

Exercise

◆ **Exercise: Constructing multiple plots**

Construct a plot which shows all the polynomial functions of the form x^n, with **n** an integer ranging from 1 to 8, over the range of **x** from -1 to 1.

◆

8.3 The FullForm of a Graphics Object Expression

The internal representation of a graphics object

First we need to understand how Mathematica represents graphics objects internally as expressions. We have already encountered this earlier when we were discussing **FullForm** and we have seen how we could use a rule to rotate and reflect a plot. We now look at this representation a little more deeply.

Because the graphics object expression is usually a large (sometimes very large) expression, Mathematica by default only shows you its full form when you ask it to. If we ask for the graphics object **p** defined in the previous section we get:

```
p
```

```
- Graphics -
```

Try generating a plot, assigning it to **p**, and then entering **FullForm[p]**. You have already seen this sort of **FullForm** output, but it is not very readable in this form.

So that we can understand the structure of this expression, we make it easier to read by using **NumberForm** to shorten the number of digits shown in the numbers (in this case we choose to display 3 digits). (Note that **NumberForm** only *displays* the numbers with less digits. Internally, they are not changed).

(In order to be able to play with the expression without Mathematica interfering due to its special way of handling graphics objects, we also need to temporarily replace the symbol **Graphics** with the symbol **graphics**. (But keep in mind that this is just a *trick* to let us *see inside* the workings of a graphics object, and the real head of a graphics object expression is **Graphics** (with a capital **G**!)))

Now let us see what the graphics object expression for **p** looks like.

```
pp = NumberForm[(p /. Graphics → graphics), 3]
```

```
graphics[
 {{Line[{{2.62 × 10^-7, 2.62 × 10^-7}, {0.255, 0.252}, {0.533,
      0.508}, {0.794, 0.713}, {1.05, 0.865}, {1.17, 0.922},
```



```
    {1.25, 0.948}, {1.31, 0.967}, {1.38, 0.982},
    {1.42, 0.988}, {1.45, 0.993}, {1.47, 0.995},
    {1.49, 0.996}, {1.5, 0.998}, {1.52, 0.999},
    {1.53, 0.999}, {1.53, 0.999}, {1.54, 1.}, {1.55, 1.},
    {1.55, 1.}, {1.56, 1.}, {1.56, 1.}, {1.57, 1.},
    {1.58, 1.}, {1.59, 1.}, {1.59, 1.}, {1.6, 1.},
    {1.6, 0.999}, {1.62, 0.999}, {1.63, 0.998},
    {1.65, 0.997}, {1.68, 0.994}, {1.72, 0.99},
    {1.78, 0.979}, {1.84, 0.965}, {1.97, 0.922},
    {2.09, 0.868}, {2.35, 0.709}, {2.6, 0.512},
    {2.87, 0.266}, {3.13, 0.0121}, {3.4, -0.259},
    {3.67, -0.502}, {3.92, -0.703}, {4.05, -0.788},
    {4.19, -0.867}, {4.33, -0.926}, {4.45, -0.966},
    {4.52, -0.981}, {4.55, -0.987}, {4.58, -0.992},
    {4.61, -0.995}, {4.64, -0.998}, {4.66, -0.999},
    {4.67, -0.999}, {4.68, -0.999}, {4.68, -1.},
    {4.69, -1.}, {4.7, -1.}, {4.71, -1.}, {4.72, -1.},
    {4.72, -1.}, {4.73, -1.}, {4.73, -1.}, {4.74, -1.},
    {4.75, -0.999}, {4.76, -0.999}, {4.77, -0.998},
    {4.79, -0.997}, {4.81, -0.996}, {4.84, -0.992},
    {4.87, -0.988}, {4.9, -0.982}, {4.97, -0.966},
    {5.04, -0.946}, {5.11, -0.922}, {5.23, -0.868},
    {5.49, -0.715}, {5.76, -0.503}, {6.02, -0.264},
    {6.27, -0.0164}, {6.28, -2.62 * 10^-7}}]]},
{PlotRange → Automatic, AspectRatio → 1/GoldenRatio,
 DisplayFunction :→ $DisplayFunction,
 ColorOutput → Automatic,
 Axes → Automatic,
 AxesOrigin → Automatic,
 PlotLabel → None,
 AxesLabel → None,
 Ticks → Automatic,
 GridLines → None,
 Prolog → {},
 Epilog → {},
 AxesStyle → Automatic,
 Background → Automatic,
 DefaultColor → Automatic,
 DefaultFont :→ $DefaultFont,
 RotateLabel → True,
 Frame → False,
 FrameStyle → Automatic,
 FrameTicks → Automatic,
 FrameLabel → None,
 PlotRegion → Automatic,
 ImageSize → Automatic,
 TextStyle :→ $TextStyle,
```


This has the simple form

```
Graphics[{{Line[{{x1, y1}, {x2, y2}, …}]}},
 {Option1 → value1, Option2 → value2, …}]
```

Or more simply, for two-dimensional graphics

```
Graphics[{graphics primitives}, {options}]
```

So, in order to understand **Graphics** objects, we only need to understand *graphics primitives* and *options*.

Extracting the points of a graphics object

Because of the adaptive sampling routine used by the **Plot** function, the coordinate points in the graphics object expression are well spaced to give a smooth curve, that is, they are more closely spaced where the curvature is highest.
Such a set of points can be useful in other circumstances. We can easily extract them from the **Graphics** object:

```
p = Plot[Sin[x],{x,0,2π}]
```



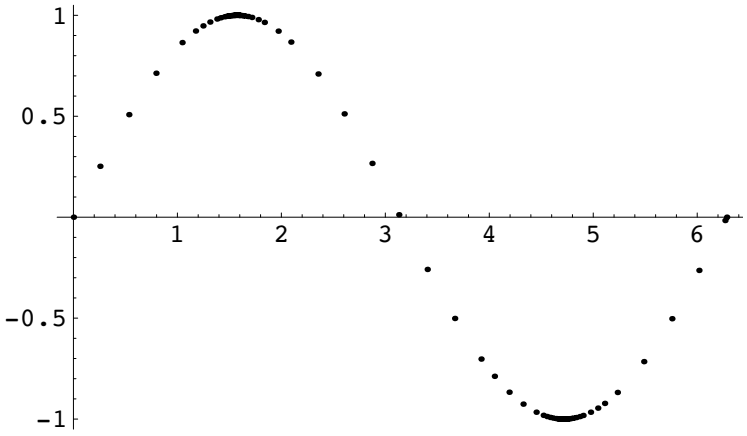
```
points = p[[1, 1, 1, 1]]
```

```
{{2.61799 × 10^-7, 2.61799 × 10^-7},
 {0.25489, 0.252139}, {0.532869, 0.508007},
 {0.793939, 0.71312}, {1.04501, 0.864929},
 {1.17413, 0.922355}, {1.24595, 0.947701},
 {1.31226, 0.966765}, {1.37966, 0.981789},
 {1.41517, 0.987915}, {1.45282, 0.993049},
 {1.46855, 0.994777}, {1.48546, 0.996361},
 {1.50009, 0.997501}, {1.51605, 0.998502},
 {1.52572, 0.998984}, {1.53469, 0.999348},
 {1.54327, 0.999621}, {1.54766, 0.999732},
 {1.55242, 0.999831}, {1.5571, 0.999906},
 {1.5614, 0.999956}, {1.56987, 1.}, {1.57756, 0.999977},
 {1.58569, 0.999889}, {1.59373, 0.999737},
 {1.59828, 0.999622}, {1.60249, 0.999498},
 {1.6184, 0.998867}, {1.63286, 0.998075},
 {1.64822, 0.997004}, {1.68008, 0.994035},
 {1.71525, 0.989585}, {1.77763, 0.978685},
 {1.8362, 0.964986}, {1.96935, 0.921622},
 {2.09144, 0.867501}, {2.35285, 0.709467},
 {2.60427, 0.511842}, {2.87186, 0.266474},
 {3.12945, 0.0121375}, {3.40323, -0.258662},
 {3.667, -0.501569}, {3.92078, -0.702701},
 {4.04933, -0.788113}, {4.19073, -0.866996},
 {4.32504, -0.925913}, {4.45069, -0.965952},
 {4.51935, -0.981426}, {4.55317, -0.987352},
 {4.58453, -0.991837}, {4.61287, -0.995052},
 {4.64304, -0.997597}, {4.66029, -0.998643},
 {4.66875, -0.999048}, {4.67652, -0.999357},
 {4.68347, -0.999582}, {4.69109, -0.999773},
 {4.69875, -0.999907}, {4.70683, -0.999985},
 {4.71547, -0.999995}, {4.72345, -0.999939},
 {4.72762, -0.999884}, {4.73221, -0.999803},
 {4.74151, -0.999576}, {4.74967, -0.999305},
 {4.75845, -0.998939}, {4.77433, -0.998082},
 {4.78997, -0.996992}, {4.80641, -0.995583},
 {4.83579, -0.992395}, {4.87, -0.987606},
 {4.90178, -0.982119}, {4.97386, -0.96601},
 {5.04347, -0.945692}, {5.10909, -0.92234},
 {5.2318, -0.86811}, {5.48592, -0.715447},
 {5.75622, -0.502911}, {6.01652, -0.263515},
 {6.26682, -0.0163632}, {6.28319, -2.61799 × 10^-7}}
```

Here, the input **p[[1,1,1,1]]** asks for the first element of the first element of the first element of the first element of **p**. Check that this is what we got!

Now, if we **ListPlot** these points, we can see how the adaptive algorithm has chosen them


```
ListPlot[points]
```




```
- Graphics -
```

Exercises

◆ **Exercise: Extracting points from the Plot**

Extract the points from the **Plot** of **Exp[-x] Cos[2π x]** where **x** runs from **0** to **4**. **ListPlot** the result.

◆ **Exercise: Modifying a plot by modifying its points**

Take the list of points above and modify the coordinates with a small random perturbation between ±5% of the coordinate. **ListPlot** the result.

◆

8.4 Two Dimensional Graphics Primitives

Defining graphics primitives

Graphics primitives are simple two dimensional geometric objects like points, lines, polygons, rectangles, circles, disks, and even text placed at a point.

To show how these work we let **pt** be a point with random coordinates between 0 and 1.

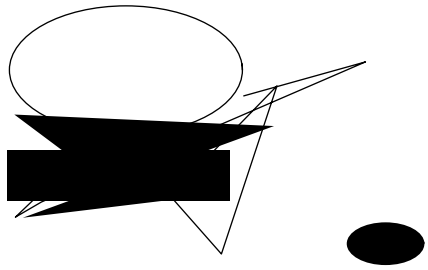
```
pt := {Random[],Random[]}
```

We can now generate some graphics objects using these primitives


```
g1 := Graphics[Point[pt]];
g2 := Graphics[Line[{pt,pt,pt,pt,pt,pt,pt,pt}]];
g3 := Graphics[Polygon[{pt,pt,pt,pt,pt}]];
g4 := Graphics[Rectangle[pt,pt]];
g5 := Graphics[Circle[pt,0.3]];
g6 := Graphics[Disk[pt,0.1]];
g7 := Graphics[Text["Hey!!",pt]];
```

Note that in order for them to be able to be rendered and displayed on the screen , we must have the object **Graphics** as the Head of each graphics expression. These can all then be shown together with the **Show** command.

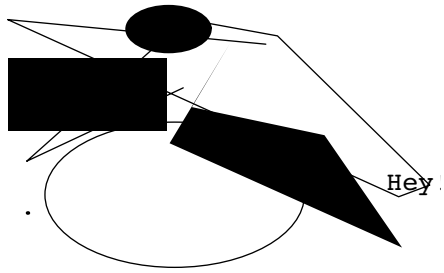
```
Show[g1,g2,g3,g4,g5,g6,g7]
```



- Graphics -

Because the points on which this display is based are random different each time **Show[g1,g2,g3,g4,g5,g6,g7]** is entered.

```
Show[g1,g2,g3,g4,g5,g6,g7]
```



- Graphics -

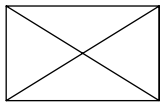
You will notice that the primitives that were supposed to be circles and disks have come out elliptical. This is because the default aspect ratio of a Graphics object is not 1:1. We can easily change this with an option. We will see how to do this next.

Exercises

◆ **Exercise: Using graphics primitives**

Create a collection of superimposed geometric objects using the graphics primitives **Line**, **Polygon**, and **Circle**.

◆ **Exercise: Drawing with `Line`**

Create an object called **`letter`** which draws an object of the form  .

◆ **Exercise: Drawing a composite object**

Create an object called **`letterToJulius`** which draws an object of the form

◆ **Exercise: Modifying the points of a plot**

Extract the points from the **`Plot`** of **`Exp[-x] Cos[2π x]`** where **`x`** runs from **`0`** to **`4`**. Replace each point with a circle of radius 0.03 centered on the point. Use **`Show`** on the resulting list of **`Graphics`** objects to show the circles. (They will come out as ellipses. Don't worry about this for now).

◆ **Exercise: Modifying plots**

Using your results from the previous exercise, construct a plot in which the circles are overlaid onto the original plot.

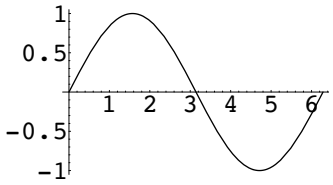
◆

8.5 Graphics Options

Extracting graphics options

If we look back at the Graphics object of the simple plot **`p`** that we made of the **`Sin`** function at the beginning of this topic we will see that apart from the graphics primitive **`Line`**, we have a list of rules as the last element. We can extract this with **`Last`**.


```
p = Plot[Sin[x],{x,0,2π}]
```




```
- Graphics -
```

```
Last[p]
```

```
{PlotRange → Automatic, AspectRatio → 1/GoldenRatio,
 DisplayFunction :→ $DisplayFunction, ColorOutput → Automatic,
 Axes → Automatic, AxesOrigin → Automatic,
 PlotLabel → None, AxesLabel → None, Ticks → Automatic,
 GridLines → None, Prolog → {}, Epilog → {},
 AxesStyle → Automatic, Background → Automatic,
 DefaultColor → Automatic, DefaultFont :→ $DefaultFont,
 RotateLabel → True, Frame → False, FrameStyle → Automatic,
 FrameTicks → Automatic, FrameLabel → None,
 PlotRegion → Automatic, ImageSize → Automatic,
 TextStyle :→ $TextStyle, FormatType :→ $FormatType}
```

These rules are called **Options**.
They control the fine detail of the way a graphics object is displayed on the screen.

Because, in generating **p**, we did not ask the **Plot** function to use any Options, the list above just applies the default values (the values on the right hand side of the →).

An easier way to look at the options to a **Graphics** object is simply to use the command **Options**.

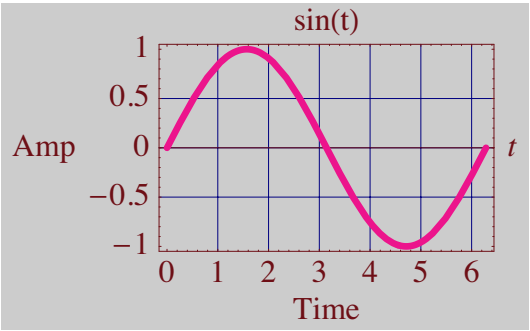
```
Options[p]
```

```
{PlotRange → Automatic, AspectRatio → 1/GoldenRatio,
 DisplayFunction :→ $DisplayFunction, ColorOutput → Automatic,
 Axes → Automatic, AxesOrigin → Automatic,
 PlotLabel → None, AxesLabel → None, Ticks → Automatic,
 GridLines → None, Prolog → {}, Epilog → {},
 AxesStyle → Automatic, Background → Automatic,
 DefaultColor → Automatic, DefaultFont :→ $DefaultFont,
 RotateLabel → True, Frame → False, FrameStyle → Automatic,
 FrameTicks → Automatic, FrameLabel → None,
 PlotRegion → Automatic, ImageSize → Automatic,
 TextStyle :→ $TextStyle, FormatType :→ $FormatType}
```


Setting graphics options

Now let us recreate a new **p** (called **pp**) with some options which we set ourselves:

```
pp = Plot[Sin[x], {x, 0, 2 π}, Frame → True,
  GridLines → Automatic, PlotLabel → "sin(t)",
  AxesLabel → {t, A}, FrameLabel → {"Time", "Amp"},
  RotateLabel → False, DefaultFont → {"Times", 12 },
  FormatType → TraditionalForm, Background → GrayLevel[0.8],
  DefaultColor → RGBColor[0.43, 0.065, 0.09], ImageSize →
   200, PlotStyle → {CMYKColor[0, 1, 0, 0], Thickness[0.02]}]
```




```
- Graphics -
```

If we look at the expression behind **pp** we will see that for the options we included above, instead of the default values, it has the values we specified.

As practice, we get Mathematica to pick out the options which have changed by using some list operations. The default ones are on the left. The new ones are on the right.

```
ip = Intersection[Last[p], Last[pp]];
TableForm[Transpose[
  {Complement[Last[p], ip], Complement[Last[pp], ip]}]]
```

AxesLabel → None	AxesLabel → {t, A}
Background → Automatic	Background → GrayLevel[0.8]
DefaultColor → Automatic	DefaultColor → RGBColor[0.43,
Frame → False	Frame → True
FrameLabel → None	FrameLabel → {Time, Amp}
GridLines → None	GridLines → Automatic
ImageSize → Automatic	ImageSize → 200
PlotLabel → None	PlotLabel → sin(t)
RotateLabel → True	RotateLabel → False
DefaultFont :→ $DefaultFont	DefaultFont :→ {Times, 12}
FormatType :→ $FormatType	FormatType :→ TraditionalForm

We will look at the **PlotStyle** option next sections where we discuss colour and line thickness .

Inspection will show the new option values have been incorporated.

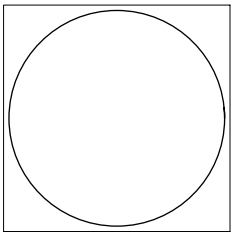
```
Options[pp]
```

$$\left\{\texttt{PlotRange} \to \texttt{Automatic}, \texttt{AspectRatio} \to \frac{1}{\texttt{GoldenRatio}},\right.$$
$$\texttt{DisplayFunction} :\to \texttt{\$DisplayFunction}, \texttt{ColorOutput} \to \texttt{Automatic},$$
$$\texttt{Axes} \to \texttt{Automatic}, \texttt{AxesOrigin} \to \texttt{Automatic},$$
$$\texttt{PlotLabel} \to \texttt{sin(t)}, \texttt{AxesLabel} \to \{\texttt{t}, \texttt{A}\}, \texttt{Ticks} \to \texttt{Automatic},$$
$$\texttt{GridLines} \to \texttt{Automatic}, \texttt{Prolog} \to \{\}, \texttt{Epilog} \to \{\},$$
$$\texttt{AxesStyle} \to \texttt{Automatic}, \texttt{Background} \to \texttt{GrayLevel}[0.8],$$
$$\texttt{DefaultColor} \to \texttt{RGBColor}[0.43, 0.065, 0.09],$$
$$\texttt{DefaultFont} :\to \{\texttt{Times}, 12\}, \texttt{RotateLabel} \to \texttt{False},$$
$$\texttt{Frame} \to \texttt{True}, \texttt{FrameStyle} \to \texttt{Automatic},$$
$$\texttt{FrameTicks} \to \texttt{Automatic}, \texttt{FrameLabel} \to \{\texttt{Time}, \texttt{Amp}\},$$
$$\texttt{PlotRegion} \to \texttt{Automatic}, \texttt{ImageSize} \to 200,$$
$$\left.\texttt{TextStyle} :\to \texttt{\$TextStyle}, \texttt{FormatType} :\to \texttt{TraditionalForm}\right\}$$

Modifying options with **Show**

We can also modify options at the stage of using the **Show** command. For example when we generated the random display of graphics primitives, we could have ensured the circle and disc came out the right shape by setting the aspect ratio to 1 by using **AspectRatio→1** as an option to the **Show** command.

```
Show[Graphics[Circle[{1, 1}, 0.3]],
 AspectRatio → 1, Frame → True, FrameTicks → None]
```



```
- Graphics -
```

Exercises

◆ **Exercise: Labeling axes**

Make a **Plot** with labeled axes.

◆ **Exercise: Coloring backgrounds**

Make a **Plot** with a coloured background.

◆ **Exercise: Adding frames and labelling them**

Make a **Plot** with a frame and frame label.

◆ **Exercise: Adding gridlines**

Make a **Plot** with GridLines, a plot label and a default font of 14 pt Times.

◆ **Exercise: Modifying density plots**

Generate a **DensityPlot** of **Sin[x y]**, with **x** and **y** ranging from **-2π** to **2π**.

Use a **PlotPoints** option of **50**.

Obtain the **Options** to **DensityPlot** and replot it with the **Mesh** removed.

◆ **Exercise: Modifying parametric plots**

Generate the three-dimensional parametric plot:

```
ParametricPlot3D[{Sin[t], Cos[t], t / 6}, {t, 0, 25}]
```

Replot it with the box and axes removed.

◆ **Exercise: Further modifying of parametric plots**

Plot the following graphic:

```
ParametricPlot3D[{Cos[t] Cos[u], Sin[t] Cos[u], Sin[u]},
 {t, 0, 2 π}, {u, -π / 2, π / 2}]
```

Replot it with the axis ticks removed, face grids added, and more plotted points.

◆

AbsoluteOptions

You can use the command **AbsoluteOptions[*graphics object*]** to give the absolute settings used by Mathematica in drawing the graphics object or plot. This may help you understand how to change the options to get what you want.

Exercise

◆ **Exercise: Exploring absolute options**

Obtain the list of absolute options for **`Plot[Sin[x],{x,0,2 π}]`**. Regenerate the Plot with some options applied to it. Obtain the new list of absolute options. Identify the changes in the new list due to the options you applied to the Plot.

◆

8.6 Style Directives

Using graphics style directives

A graphics style directive causes a change in the *style* in which the primitives are rendered. They change the colour, the size of points, the thickness of lines, or make the lines dashed.

For a number of objects, each with different style directives you can use the form

```
Show[Graphics[{{style directives, object},
   {style directives, object}, …}], overall options]
```

If one set of style directives applies to several different objects, you can use the form

```
Show[Graphics[{style directives, object, object, …}],
 overall options]
```

Or you can use combinations of these two forms.

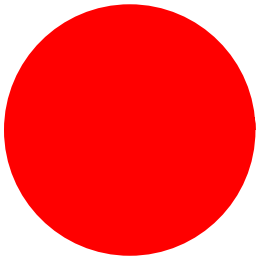
We will now explore some of the more common style directives.

Color

You can color a **`Graphics`** object by using the graphics directive **`RGBColor`**. (Note carefully the spelling of **`Color`**.

For example we can display a red disk by entering

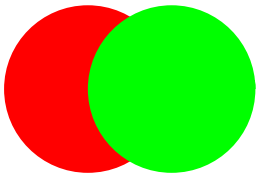

```
Show[Graphics[{RGBColor[1, 0, 0], Disk[{0, 0}, 1]},
  AspectRatio → Automatic, ImageSize → 100]]
```



```
- Graphics -
```

We can display a red disk and a green disk by writing a different colour directive with each disk object.

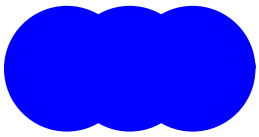
```
Show[Graphics[{{RGBColor[1, 0, 0], Disk[{0, 0}, 1]},
   {RGBColor[0, 1, 0], Disk[{1, 0}, 1]}}],
 AspectRatio → Automatic, ImageSize → 100]
```



```
- Graphics -
```

Or, we could have displayed three blue disks with the one colour directive

```
Show[Graphics[{RGBColor[0, 0, 1],
   Disk[{0, 0}, 1], Disk[{1, 0}, 1], Disk[{2, 0}, 1]}],
 AspectRatio → Automatic, ImageSize → 100]
```

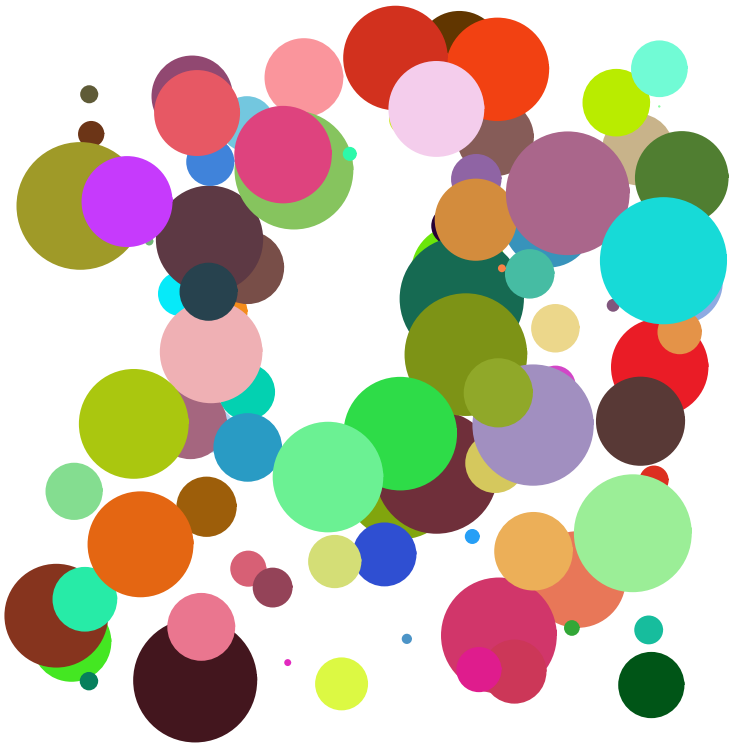


```
- Graphics -
```

Note that the disks must now be drawn smaller to keep the *overall* **ImageSize** option to the same value of **100**.

By making a table of graphics objects like this, we can splatter the page with coloured disks.


```
Show[Graphics[Table[{RGBColor[Random[], Random[], Random[]],
    Disk[{Random[], Random[]}, Random[] / 10]}, {100}]],
 AspectRatio → Automatic]
```



```
- Graphics -
```

Selecting your colours

The best way to select your colours is to use the *Color Selector*. Go to the menu Input: Color Selector, select your colour and then click OK. Your **RGBColor** values will be pasted into your notebook wherever your cursor is.

GrayLevel

GrayLevel[*lightness*] (N*ote carefully the spelling!*) gives a gray colour, with the argument *lightness* going from **0** (black) to **1** (white).

Exercises

◆ **Exercise: Showing random arrays of colored objects**

Show a random array of randomly coloured rectangles.

◆ **Exercise: Using color styles in plots**

Plot a graph using the **Background**, **DefaultColor**, and **PlotStyle** options with values of these options being **RGBColor**s that you have selected using the Color Selector.

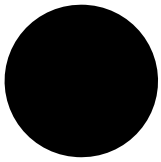
◆

Point size

You can change the point size of point by using the **Graphics** directive **PointSize**.

For example, to show a point of size **0.2** (that is, of size **20%** of the width of the picture) we could enter

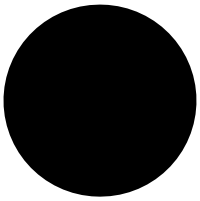
```
Show[Graphics[{PointSize[0.2], Point[{0, 0}]}]]
```



```
- Graphics -
```

If you want the point to have a fixed absolute dimension, you use **AbsolutePointSize** instead.


```
Show[Graphics[{AbsolutePointSize[72], Point[{0, 0}]}]]
```



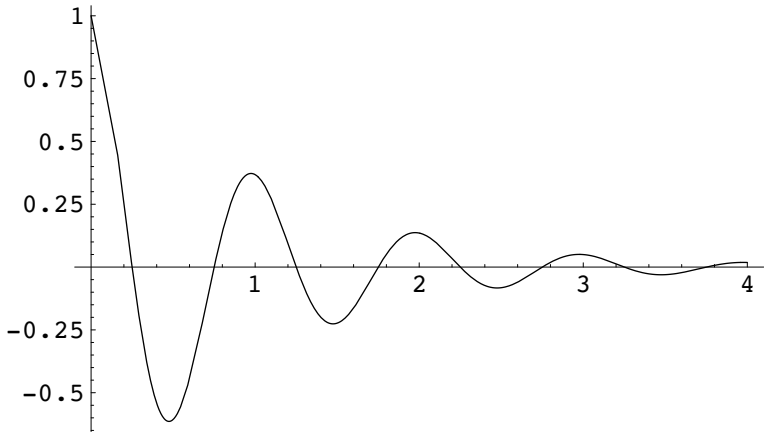
```
- Graphics -
```

Absolute graphics dimensions are measured in *screen pixels*. A screen pixel of a medium resolution computer screen is very roughly equal to a printer's *point*. There are 72 points in an inch (or, very approximately, 3 per millimetre).

◆ **Example**

Suppose we had a plot generated by the Plot function, and we wanted to turn the points generated by the Plot function into a series of points that you could change the size of.

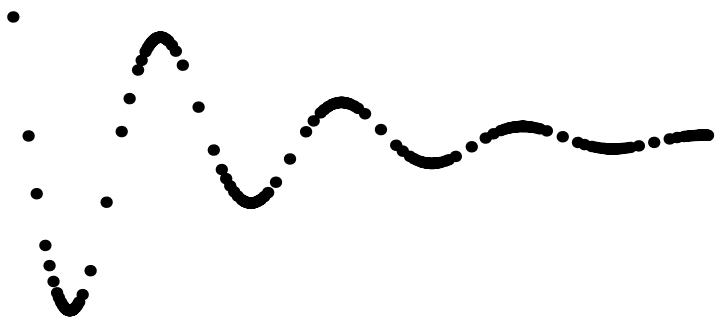
```
pe = Plot[Exp[-x] Cos[2 π x], {x, 0, 4}]
```




```
- Graphics -
```



```
Show[pe[[1, 1, 1, 1]] /.
  {x_, y_} :> {Graphics[{PointSize[0.015], Point[{x, y}]}]}]
```



```
- Graphics -
```

Exercise

◆ **Exercise: Exploring point size directives**

Take the plot from the example above and **Show** its points randomly coloured with absolute size 3 mm.

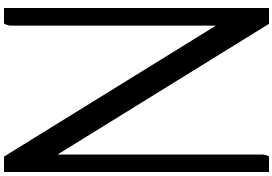
◆

Thickness

The **Thickness** graphics directive applies to lines. Just as for **PointSize**, there is also an **AbsoluteThickness**.

You can apply **Thickness** or **AbsoluteThickness** to a **Line** object:

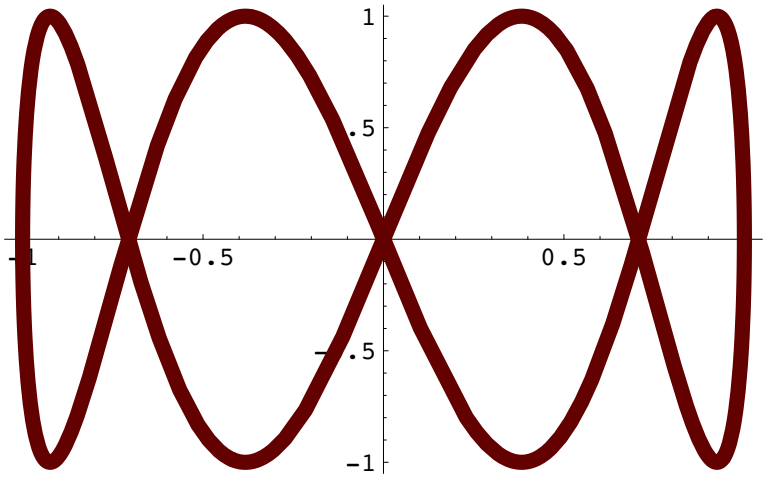
```
Show[Graphics[{Thickness[0.1],
   Line[{{0, 1}, {1, 1}, {0, 0}, {1, 0}}]}], ImageSize → 100]
```



```
- Graphics -
```

Or, you can apply it directly to a **Plot**, using the **PlotStyle** option


```
ParametricPlot[{Sin[t], Sin[4 t]},
 {t, 0, 2 π}, PlotStyle → {Thickness[0.02],
   RGBColor[0.38999, 0.0117037, 0.0243839]}]
```




```
- Graphics -
```

Exercises

◆ **Exercise: Modifying the thickness of a line**

Make a **Plot** with a line thickness of 5 points.

◆ **Exercise: Modifying the dashed form of a line**

Explore the graphics directives **Dashing** and **AbsoluteDashing**. By researching their syntax, and then using them in the **PlotStyle** option to a **Plot**.

◆

8.7 Controlling Graphic Output

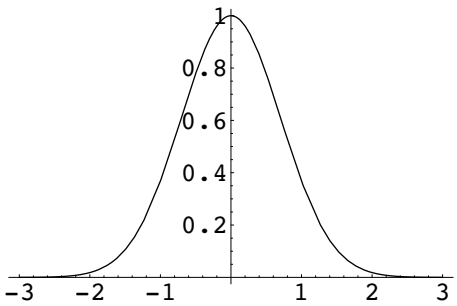
Graphic displays as side-effects

When writing a function to combine two or more plots, it is useful to be able to suppress the automatic side-effect plotting of each one before the combination is made. Normally you can suppress the display of output by using the semi-colon. For example, if we want to generate π to 1000 places but *not* to display it we can enter

```
N[π, 1000];
```


However, if we try to generate a graphics object with a **Plot** command, and a semicolon, we *still get the plot displayed*.

```
Plot[e^-x^2, {x, -3, 3}];
```



The key to understanding this behaviour is to note that the *display of the output* was in fact suppressed, but the plot is not the output, so it was not suppressed. Since the *display of the plot* is a *side-effect* of the calculation, it was not suppressed by the semi-colon.

- We suppress a display of the plot by using the option

```
DisplayFunction → Identity
```

- We force a display of a plot by using the option

```
DisplayFunction → $DisplayFunction
```

◆ **Example**

As an example we generate 2 graphics objects using **Plot**, but suppress their display

```
p1 = Plot[e^-x^2, {x, -3, 3}, DisplayFunction → Identity]
```

```
- Graphics -
```

```
p2 = Plot[-e^-x^2, {x, -3, 3}, DisplayFunction → Identity]
```

```
- Graphics -
```

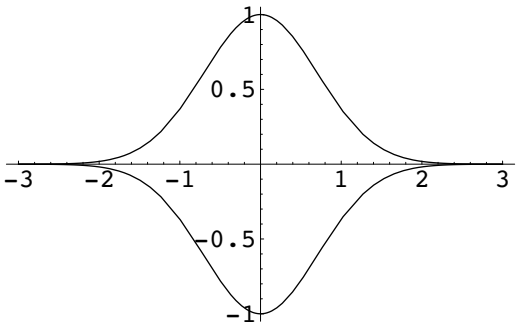
Applying **Show** will combine them, *but not show them*, because we have explicitly included the option **DisplayFunction → Identity**.

```
Show[p1, p2]
```

```
- Graphics -
```

In order to show the combined object we need to include the option **DisplayFunction → $DisplayFunction**.

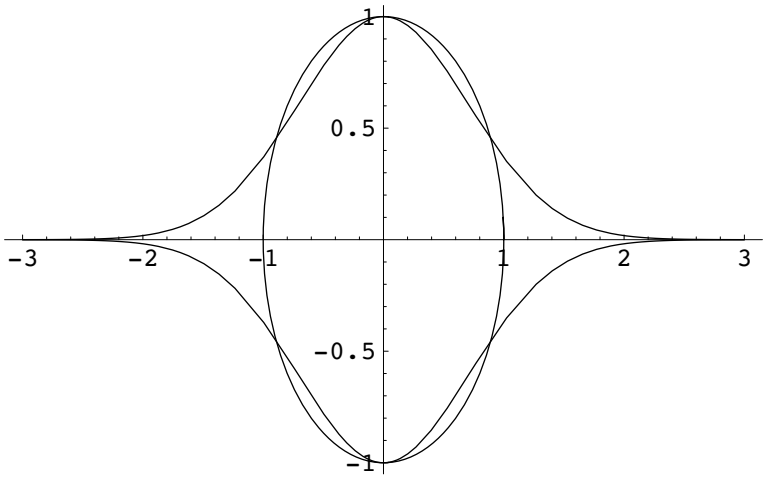

```
p3 = Show[p1, p2, DisplayFunction → $DisplayFunction]
```




```
- Graphics -
```

If we want to combine a circle with this we could use **Show** again

```
Show[p3, Graphics[Circle[{0, 0}, 1]]]
```




```
- Graphics -
```

Exercise

◆ **Exercise: Suppressing intermediate displays**

Construct a plot which shows all the polynomial functions of the form x^n, with **n** an integer ranging from 1 to 8, over the range of **x** from -1 to 1. Construct it so that only the final plot, and none of the intermediate plots are displayed.

◆

8.8 Graphics Packages

Exploring graphics packages

There is a lot of graphics capability in the graphics packages that come with Mathematica, for example the package "Graphics", or in the packages which you can find on *MathSource*, or which are provided on disc with some of the Mathematica books.

For example *Mathematica Graphics* (by Tom Wickham-Jones) has a package for labeling contour plots.

Load a Graphics package. *Note carefully the use of the* ***backquote*** **`**. (This is *not* a quote. It is found on the same key as the tilde ~). For example **Graphics`FilledPlot`**

```
<< Graphics`FilledPlot`
```

To find out how to use the functions in this package, go to the Help Browser (Help...) under the Help menu. Click Add-Ons, then Standard Packages > Graphics. Then choose from the list you see in the third scroll box.

Note carefully that *you* ***must*** *load the package* ***before*** *entering any functions in the package*. If by chance you don't, you can recover from the confusion by using **Remove** to remove the offending function names you entered too soon. (A simpler way may be to quit Mathematica and start a new session).

Exercises

◆ **Exercise: Loading a graphics package**

Load the Graphics package **Graphics`FilledPlot`**. Try out the **FilledPlot** function for plotting a graph filled to the axis.

◆ **Exercise: Exploring graphics packages**

Look at the information on some of the functions described and try them out.

◆

8.9 Graphics Arrays

Showing an array of graphics images

You can show an array of graphics images, that you have generated using the command **GraphicsArray**. The syntax is

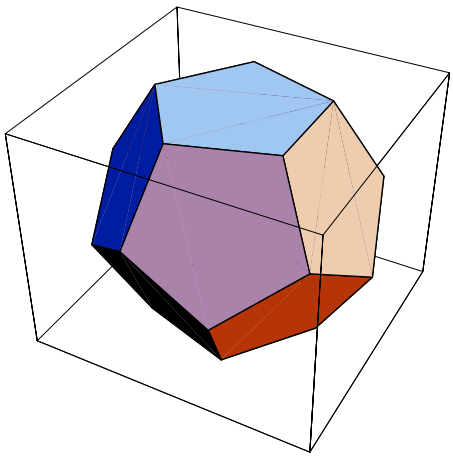
```
Show[GraphicsArray[{{plot1, plot2, …}, {plot3, plot4, …}}]]
```

For example, suppose you were interested in displaying various polyhedra. You can access most of them from the **Graphics`Polyhedra`** package. First load the package

```
<< Graphics`Polyhedra`
```

You can show them individually

```
p1 = Show[Graphics3D[Dodecahedron[]]]
```



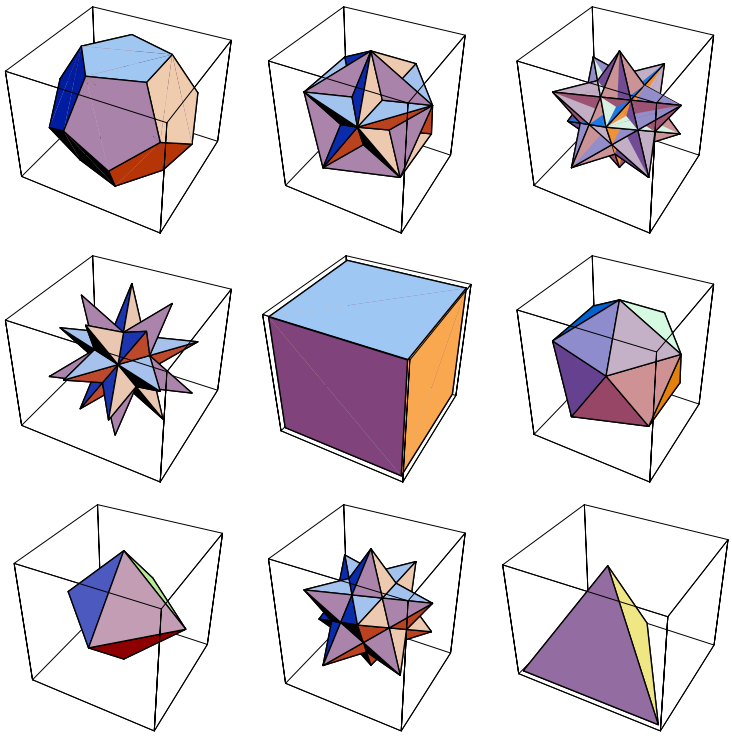
```
- Graphics3D -
```

Or in a **GraphicsArray**. First collect the polyhedra into the array configuration you want

```
poly = {{Graphics3D[Dodecahedron[]], Graphics3D[
      GreatDodecahedron[]], Graphics3D[GreatIcosahedron[]]},
    {Graphics3D[GreatStellatedDodecahedron[]],
     Graphics3D[Hexahedron[]], Graphics3D[Icosahedron[]]},
    {Graphics3D[Octahedron[]],
     Graphics3D[SmallStellatedDodecahedron[]],
     Graphics3D[Tetrahedron[]]}};
```



```
Show[GraphicsArray[poly]]
```



```
- GraphicsArray -
```

Exercise

◆ **Exercise: Exploring graphics arrays**

Plot four separate graphs and combine them into 2×2 graphics array.

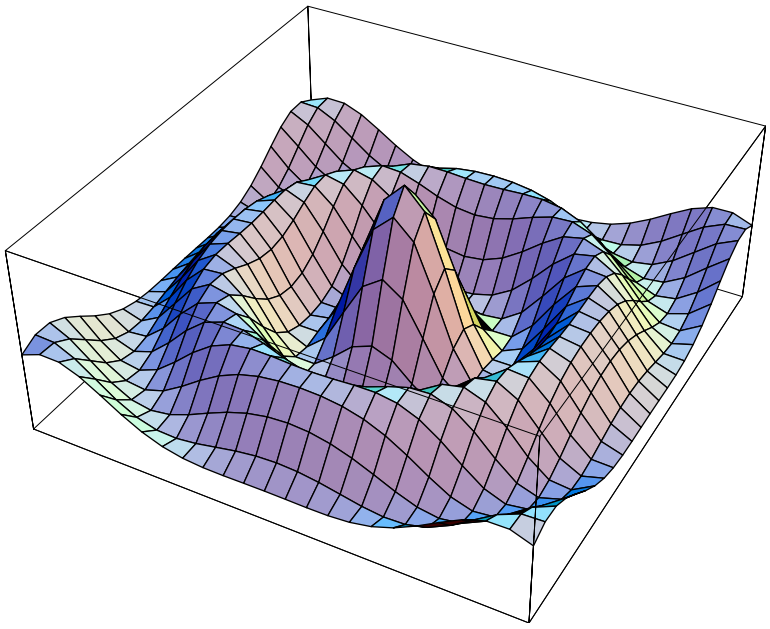
8.10 Animation

Generating an animation sequence

To animate a sequence of graphics, all you need to do is to generate them one after the other in different cells. Then select the cells and choose **Animate Selected Graphics** from the **Cell** menu.

For example, create a list of **Graphics** objects. Each one will be displayed in a cell as it is created (provided you do not suppress them). You can collapse the bracket that contains all these cells for neatness and just select the outer enclosing bracket to animate them via the menu or keyboard shortcut.


```
Table[Plot3D[BesselJ[0, Sqrt[x^2 + y^2] + t],
  {x, -10, 10}, {y, -10, 10}, Axes → False,
  PlotRange → {-0.5, 1.0}, PlotPoints → 25], {t, 0, 8}]
```



```
{- SurfaceGraphics -, - SurfaceGraphics -, - SurfaceGraphics -,
 - SurfaceGraphics -, - SurfaceGraphics -, - SurfaceGraphics -,
 - SurfaceGraphics -, - SurfaceGraphics -, - SurfaceGraphics -}
```

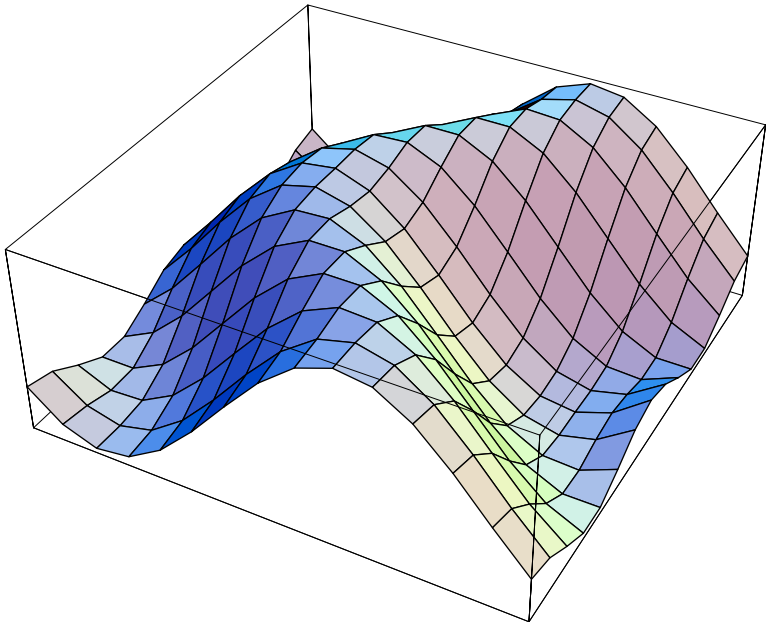
Generating a spin sequence

There is another command which creates animation frames (a frame is simply a graphics object in a cell), and which enables you to spin a graphic. First load the **<<Graphics`Animation`** package.

```
<< Graphics`Animation`
```

Then generate a graphic.

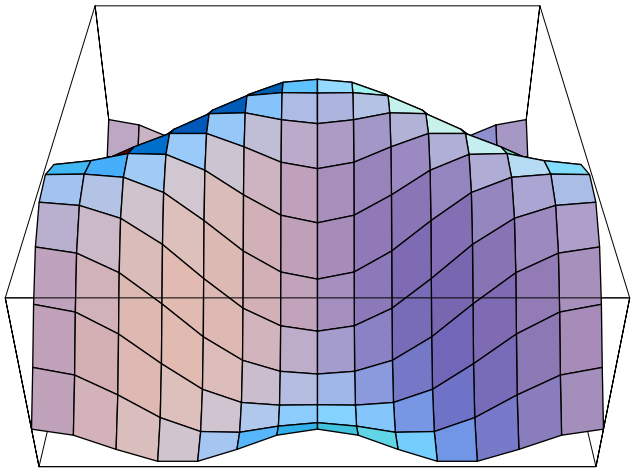

```
gg =
 Plot3D[Cos[x + Cos[y]], {x, -π, π}, {y, -π, π}, Axes → False]
```



```
- SurfaceGraphics -
```

Now generate some frames for this with **SpinShow**. You can then collapse the cells and animate them as before.

```
SpinShow[gg, Frames → 12]
```



Exercises

◆ **Exercise: Spinning a graphic**

Load the package **<<Graphics`Animation`**. Generate a graphic and apply **SpinShow** to it. Animate the resulting graphics sequence of cells.

◆ **Exercise: Animating a graphic**

Generate a list of graphics objects with the **Table** function. Animate the resulting sequence of cells.

8.11 Problems

Problem 1: Modifying the points of a plot

Write a function called **randomPlot[f,{x,a,b},e]** which acts like the **Plot** function, but adds a percentage error **e** to the points in the plot. (See the first exercise in the section.)

◆

Problem 2: Further modification of the points of a plot

Design a function called **circlePoints** which takes a **Plot** and a diameter as arguments and generates plot in which the plotted points are surrounded by circles of the given diameter. (See the relevant exercises above).

◆

Problem 3: Generating random graphics displays

Create the following graphic:

```
DensityPlot[Sin[x / Cos[y]],
 {x, -π, π}, {y, -π, π}, ColorFunction → Hue,
 PlotPoints → 40, Mesh → False, Frame → False]
```

Create a random array of four rectangles each filled with this graphic. *Hint*: Check out the syntax for **Rectangle**.

Create a function called **splatter[]** which splatters **4** of these rectangles over the page. Use a **Module** to combine your two previous results. Write your function so that the original graphic is not displayed. Observe that some rectangles might obscure others.

◆

Problem 4: Picking off graphics coordinates

Generate a **ContourPlot** of **Sin[x y]**, with **x** and **y** ranging from **-π** to **π**. Use a **PlotPoints** option of **50**. And a **Contours** option of 4 (to generate 4 contours).

Go to the Input menu and select Get Graphics Coordinates. Pick off the points from the innermost contour, and use **ListPlot** to plot them joined with a line.

◆

Problem 5: Using color directives

A short piece of code producing a three dimensional graphic out of a table of **Cuboid** objects is

```
Show[Graphics3D[
     Table[{RGBColor[i,j,k],Cuboid[10{i,j,k}]},
           {i,0,1,0.2},{j,0,1,0.2},{k,0,1,0.2}]],
    Lighting->False,Boxed->False]
```

Show the graphic. Then modify the code to make the colours of the cuboids random.

Write a function called **cubeStack** which generates a stack of cubes of a different size depending on the value of its single argument.

◆

Problem 6: Viewing plots from different angles

Plot the following function

```
Plot3D[Sin[x] Cos[y], {x, -2 π, 2 π},
 {y, -2 π, 2 π}, PlotPoints → 100,
 PlotRange → {-1, 0.5}, Mesh → False, FaceGrids → All]
```

Use the 3D ViewPoint Selector in the Input menu find the values of a **ViewPoint** option which will enable the viewing of the underneath of the surface.

Generate the plot viewed from underneath.

◆

Problem 7: Generating graphics arrays

Create a Graphics array of the function $e^{-\frac{a t}{8}}$ `Cos[b t]` with `t` from `0` to `6`. The parameters `a` ranges from `1` to `4` and `b` ranges from `1` to `3`. Use the `Table` function to generate the plots of which the array is composed, but suppress their individual display. To make the individual plots more readable you may need to resize the font with the `DefaultFont` option, and force all of the plot to be displayed with the `PlotRange` option. Expand the size of the overall GraphicsArray with the ImageSize option.

◆

Problem 8: A function which generates a graphics array

Write a function called `plotArray` which generates a graphics array as in the previous problem, but which allows the maximum value of `t` which is plotted to be varied (that is, the maximum value is the argument to the function).

Test out your function.

◆

