First Course in Scientific Computing
Table of Contents

Matrix Operations

Eigenvalues, Eigenvectors, Linear Equations

Generating Fortran & C Code

Mathematica standard Library Functions

Mathematica Packages

Chapter 4: Visualization, Abstract Data Types; Electric Fields

4.1 Why Visualization?

One of the most rewarding uses of computers is *visualizing* the results of calculations. This is done with 2-D and 3-D plots (especially with colored surfaces), with contour maps, and with animations. These types of visualization can be breathtakingly beautiful and often provide deep insight into a problem by letting you see and ``handle'' the functions with which you are working. Visualization also assists the program debugging process, the development of physical and mathematical intuition, and the all-around enjoyment of your work. Some of the reasons for this may arise from the fact that some large fraction (~50%) of our brain gets involved in visual processing, and if you can use this extra brain power in your scientific work, then you have extended what was otherwise possible with solely logical abilities.

Traditionally, visualization of a scientific problem was the last step in problem solving. After studying table of numbers for hours and gaining confidence that they are right, a scientist might then go to the trouble of making a bunch of 2-D plots to examine various aspects of data. Well, in present times computational scientists have demonstrated how much there is to be gained by going beyond 2-D plots. Now it is regular practice to use surface plots, volume rendering (dicing and slicing), and animations (movies). In this chapter, we use some of these techniques within the context of visualizing the electric potential around charges.

4.2 Problem: Stable Points in Electric Fields

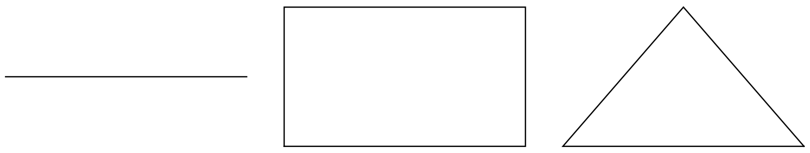


Figure 4.1 Static configuration for two, three, and four charges (the charges are at the corners).

You are given some simple configurations of two, three, and four charge systems, as shown in the Fig. 4.1. The two charges are fixed on a line at coordinates (1,0), (-1,0); the three charges are fixed to the corners of an equilateral triangle at coordinates (0,1), ($\sqrt{3}$ /2, -1/2), (-$\sqrt{3}$ /2, -1/2); and the four charges are fixed to the corners of a square at coordinates (1,1), (1,-1), (-1,-1), (-1,1). The origin is at the center of each geometric figure. Your **problem** is to determine the electrical potential at the arbitrary point (x,y) and see if there might be some points in space at which we can place a charge that is free to move and have it remain there even if perturbed. (For the equivalent gravitational problem these stable points are known as *Lagrange* points and are the location of asteroids for the earth-sun system.)

4.3 Theory: Stability Criteria for Potential Energy

Coulomb's law tells us that if we have a charge q at the origin, then the electric field **E** (the force per unit charge) at a distance r from that charge is

$$\boldsymbol{E}(r) = \frac{k_e\, q}{r^2}\, \hat{\boldsymbol{r}}$$

where $\hat{\mathbf{r}}$ is a unit vector in the radial **(r)** direction. Here $k_e = 8.9875\ 10^9\,\mathrm{N}\ m^2/C^2$ is Coulomb's constant in SI units, and the electric force **E** is directed radially away from the charge. Because **E** is a vector, the electric force field about a charge is a vector field with both magnitude and direction at each point. However, no information is lost, and it's much simpler, if, instead of the electric field **E**, we consider the electric *potential* field

$$V(r) = \frac{k_e\, q}{r} = \frac{q}{r}$$

In the second form of this equation we have left off the electric constant `ke` for simplicity; since this affects just the magnitudes of the graphs and not their shapes, it will not change the conclusions we draw. We see that *V(r)* falls off less rapidly than **E**(r) and is a scalar, that is, has no direction associated with it.

Our problem requires us to determine the potentials for two- and three-charge systems, and then to look for stable points in these potentials. To determine the potential for two charges, we use Pythagorean's theorem to determine the distance to the charges,

$$r_1 = \sqrt{(x-a)^2 + y^2}\,,\quad r_2 = \sqrt{(x+a)^2 + y^2}$$

and then just add up the potentials from the individual charges:

$$V_2(x, y) = \frac{\mathrm{q1}}{\sqrt{(x-a)^2 + y^2}} + \frac{\mathrm{q2}}{\sqrt{(x+a)^2 + y^2}}$$

For three charges at the corners of the equilateral triangle, we know the coordinates are (0,a), (a cos θ, -a sin θ), (-a cos θ, -a sin θ), where $\theta = 30$ degrees. Again we use Pythagorean's theorem and add the potentials from the individual charges to obtain

$$V_3(x, y) = \frac{q_1}{\sqrt{x^2 + (y-a)^2}} + \frac{q_2}{\sqrt{(x - a\cos\theta)^2 + (y + a\sin\theta)^2}} + \frac{q_3}{\sqrt{(x + a\cos\theta)^2 + (y + a\sin\theta)^2}}$$

These equations for the electric potentials are what we wish to visualize. To make them simpler to visualize, we set $a = 1$ and substitute for θ:

$$V_1(x, y) = \frac{q_1}{\sqrt{x^2 + y^2}}, \qquad V_2(x, y) = \frac{q_1}{\sqrt{(x-1)^2 + y^2}} + \frac{q_2}{\sqrt{(x+1)^2 + y^2}}$$

$$V_3(x, y) = \frac{q_1}{\sqrt{x^2 + (y-1)^2}} + \frac{q_2}{\sqrt{\left(x - \frac{\sqrt{3}}{2}\right)^2 + \left(y + \frac{1}{2}\right)^2}} + \frac{q_3}{\sqrt{\left(x + \frac{\sqrt{3}}{2}\right)^2 + \left(y + \frac{1}{2}\right)^2}}$$

Owing to its two dimensional nature, a purely mathematical solution for the equilibrium points in these potential gets complicated. Instead, we will solve it graphically and rely on our intuitive understanding of how balls roll under the action of gravity. Specifically, we know that a ball released on a surface rolls downhill, and that if the ball is placed in a concave depression, it will remain there. Because the gravitational potential near the Earth's surface is proportional to height, our description of the ball on a surface is equivalent to a description of how a particle behaves in a potential energy field. It therefore follows that charges will ``roll down'' the electric potential surface, and will find a stable position at the concave minimum of the potential. So our problem translates into drawing pictures of the electric potential surfaces and looking for minima at the bottom of hills.

4.4 Basic 2-D Plots: Plot

Before we get to Maple's plotting commands, let us examine some general principles. First, keep in mind that the point of visualization is to make the science clearer and to better communicate your work to others. So it follows that when you produce a figure you should look at it and think if there are some better choices of units, ranges of axes colors, style, *et cetera* that might get the message across better and provide better insight. Taking into account that we are dealing with the complexity of human perception and cognition, there may not be one definite way to do things, and some trial and error is necessary to see what looks best.

Our general recommendation for visualization is to make each figure as clear, informative and self-explanatory as possible. This means labels for various curves and data points, a title, and labels on the axes. We know, you are thinking that this is really a lot of work for a lousy assignment or report, and that you do not need all those time-consuming extras to comprehend what is going on. Yet the more often you do it, the quicker and better you get at it, and the more useful will your work be to others (and yourself in the future).

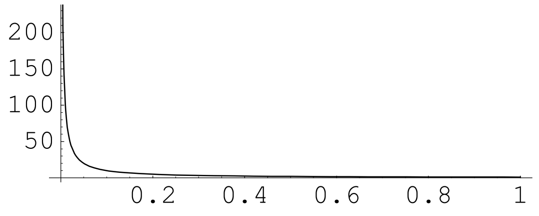
The convention when plotting is to have the independent variable, say *x*, along the abscissa (horizontal axis), and the dependent variable, say *y=f(x)*, along the ordinate. (Remember that your mouth spreads horizontally across when you say ``abscissa'', and that it puckers vertically up when you say ``ordinate''.) If you have trouble deciding which variable is independent, think of an experiment in which you measure the position or velocity of a ball as a function of time. Because you are free to pick the times at which you make the measurement, time is an independent variable. However, once you have chosen the time, nature picks what the position of the ball is at that particular time, so position and velocity are dependent variables.

Mathematica excels at easily producing graphs of all sorts, and indeed, visualization is one of the most valuable aspects of *Mathematica*. While we will discuss and give examples of a number of possible plots, *Mathematica* offers you more options than we discuss, and we recommend you browse the `Help` pages to create just the graph you want. Unlike Maple, where you need to load the graphics tools before you use them, the basic graphics tools in *Mathematica* are contained in the standard package and can be used without loading. However, we will need to load specific packages later as we go beyond the basics. We will first make a simple plot and then embellish it with things like labels and colors. Let us start by looking at the electric potential for a single charge (using our natural units) at the origin as defined by a function *V(r)*.

```
In[1]:= V[r_] := 1 / r
        (*  Remember underscore needed to define function, not when used  *)
        ?V
        Null
       Global`V
```

V[r_] := $\frac{1}{r}$

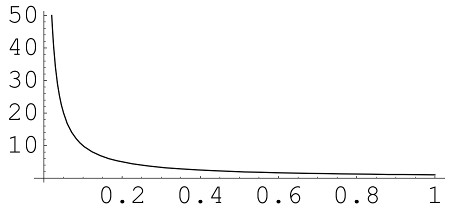
```
In[4]:= Plot[V[r], {r, 0, 1}]         (*  Plot previously defined function  *)
```



```
Out[4]= - Graphics -
```

You will observe from the figure that the second argument to the `Plot` command is a list that gives the range of values for the abscissa (*r* in this case). Our interest is really for *r* between 0 and infinity, but this does not produce such a useful result since we mainly see the repulsive peak at the origin. So we can get a more revealing plot by not letting *r* get quite so close to the origin:

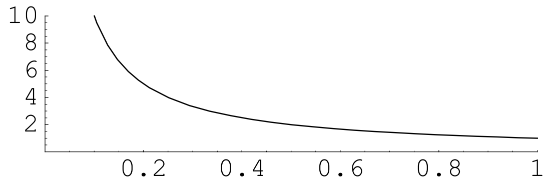
```
In[5]:= Plot[V[r], {r, 1 / 50, 1}]
```



```
Out[5]= - Graphics -
```

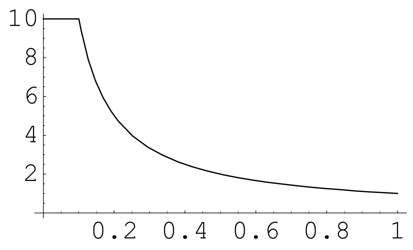
However, the plot has left out the *r=0* part of the graph, and so does not fully convey the image that the potential is infinite there. We can tailor our plot more to our liking by giving some limits to the ordinate (also works if called generic *y*):

```
In[6]:= Plot[V[r], {r, 0, 1}, PlotRange → {{0, 1}, {0, 10}}]
        (*  Limit y range (cute,huh!)  *)
```



```
Out[6]= - Graphics -
```

```
In[7]:= Plot[Min[V[r], 10], {r, 0, 1}]
        (*  Keep ordinate <10 another way  *)
```



```
Out[7]= - Graphics -
```

As an alternative, you can tell *Mathematica* that you want to see the full dependence of the potential from *r=0* to ∞, although you may lose some details:

```
In[8]:= Plot[V[r], {r, 0, Infinity}]

        Plot::plln :
         Limiting value ∞ in {r, 0, ∞} is not a machine-size real number.

Out[8]= Plot[V[r], {r, 0, ∞}]
```

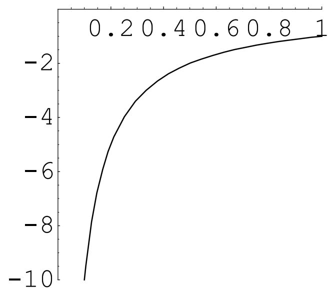
Well, it didn't like that. You can try leaving off the range for the abscissa as well, to see how smart *Mathematica* really is:

```
In[9]:= Plot[V[r]]

        Plot::argmu : Plot called with 1 argument; 2 or more arguments are expected.

Out[9]= Plot[V[r]]
```

You see that because *Mathematica* was not given a range of *r* values to plot, it throws an error since it expects two or more arguments. Pretty smart!

The plot above shows the basic physics. If we view *V(r)* as an equivalent gravitational potential, a small positive charge (mass) placed near the fixed positive charge will be repelled (roll downhill) out to infinity. There are not any locations where a charge remains at rest in equilibrium. If we had fixed a negative charge at the origin, the potential would have the opposite sign:

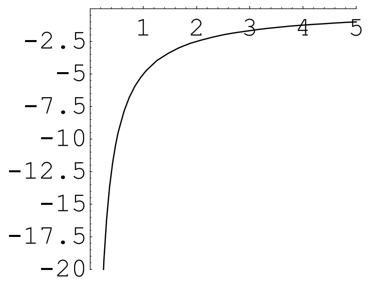
```
In[10]:= Plot[-V[r], {r, 0, 1}, PlotRange -> {{0, 1}, {0, -10}}]
```



```
Out[10]= - Graphics -
```

This shows that, regardless of where we place it, our positive test charge will fall into the hole at the origin. As we have just seen by placing a minus sign in front of the first argument to the command, it is allowable to have the argument be an expression and not just a function:

```
In[11]:= Plot[-5 / r, {r, 0, 5}, PlotRange → {{0, 5}, {0, -20}}]
         (*  Plot explicit expression  *)
```



```
Out[11]= - Graphics -
```

In summary, the first argument to the `Plot` command is the name of the function or expression to be plotted along the ordinate, that is, the dependent variable. The second argument is the range of values for abscissa, that is, the independent variable. The range is specified in a list containing the name of the independent variable, the minimum value, and the maximum value, *e.g.*, `{x, -10, 10}`. In order to place a range on the ordinate, the `PlotRange` option must be used. Options are rules indicated by an arrow ->, with the arguments enclosed in curly braces.

After evaluating the command you may notice the `-Graphics-` description following the produced graph. This indicates that the plot is a ``graphics'' object, which can be manipulated. The description may be suppressed by placing a semicolon ; to the right of the `Plot` command. Try putting a semicolon after the last `Plot` command and re-evaluate the cell. We will do this for the rest of our examples.

Before we get on to embellishing the `Plot` command, let us have some fun with the pretty graph you just produced:

- Click on the graph to select it.

- Note how a box is formed around the selected graph and that there are dark little nodes at the corners and in the middle of the sides.

- Use your mouse to resize the graph by grabbing one of the nodes, and then dragging it with the mouse button still depressed. Notice that when a node is selected, a little two-sided arrow appears to show you the direction in which the frame can be resized. The resizing can be done diagonally along the corners or horizontally and vertically along the edges. [If you find that you can only resize diagonally, with both sides having the same ratio, then the `AspectRatioFixed` option under `Graphics Options` needs to be changed to `False`. To do this, make sure your graph is selected, then open the `Format` menu and choose `Option Inspector...`. A separate window will appear where you can click on the arrow next to `Graphics Options`, then `Image Bounding Box`. You will then see the property that needs to be changed in order to resize your graph any way you wish.]

- Select the graph, copy it (it's placed on the ``clipboard''), and then paste it back to the notebook so that now you have two graphs.

- Make one of your graphs into a long, narrow one, and the other into a high, tall one. Notice how the tall one emphasizes the variation in the magnitude of *V(r)*, while the long one emphasizes the range of *r* values to which *V(r)* extends. Both are perfectly legitimate ways to view a function. One emphasizes the singular nature near the origin, the other the long range of the potential.

- Another way to view a function, especially one that has orders of magnitude variation in value, is with a semilog plot or a log-log plot. (Note, however, that you cannot take the log of 0). In the Input cell below, use the `Log[10, z]` function to see how a semilog plot changes the appearance of the same *f(x)* we have been viewing. Follow the instructions in the comment fields below:

```
In[12]:= (*  Repeat plot with Log[10,V (r)]  *)
```

Next try the explicit semilog plot function, `LogPlot`.

■ 4.4.1 Loading Graphics Package

Before attempting to use that function, we need to load the `Graphics` package, which contains it. For many calculations the standard *Mathematica* package is sufficient, but often you may need more specialized functions and procedures that are not loaded with *Mathematica*, but are found in various *Mathematica* packages. They are loaded with the << command:

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

where `Graphics` is the name of the package (folder) in *Mathematica*'s `StandardPackages`, and `Graphics.m` is the name of the file being loaded. You may also load this file from a specific location (if you know the absolute path of the package), using the `Get` command:

```
In[2]:= (* Get["Graphics.m",
          Path→{"/usr/local/mathematica/AddOns/StandardPackages/Graphics/" }]  *)
```

Notice how both arguments are enclosed in quotes, and the `Path` option in curly braces.

Once a package is loaded you can get a list of the functions it contains by using the `Names` command:

```
In[2]:= Names["Graphics`Graphics`*"]

Out[2]= {BarChart, BarEdges, BarEdgeStyle, BarGroupSpacing, BarLabels, BarOrientation,
         BarSpacing, BarStyle, BarValues, DisplayTogether, DisplayTogetherArray,
         ErrorListPlot, GeneralizedBarChart, Histogram, LabeledListPlot,
         LinearLogListPlot, LinearLogPlot, LinearScale, ListAndCurvePlot,
         LogGridMajor, LogGridMinor, LogLinearListPlot, LogLinearPlot, LogListPlot,
         LogLogListPlot, LogLogPlot, LogPlot, LogScale, PercentileBarChart,
         PieChart, PieExploded, PieLabels, PieLineStyle, PieStyle, PiScale,
         PolarListPlot, PolarPlot, ScaledListPlot, ScaledPlot, SkewGraphics,
         StackedBarChart, TextListPlot, TransformGraphics, UnitScale}
```

An easy mistake is trying to use a command before its package has been loaded, then loading the package, and then trying to use the command again. This leads to an error message due to the way the command was initially entered into the symbol table. You need issue the `Remove[`*`command name`*`]` command before loading the package. For example, enter:

```
In[3]:= Show[Graphics3D[Cylinder[0.5, 0.5]]];

        Graphics3D::gprim : Cylinder[0.5, 0.5] was encountered
           where a Graphics3D primitive or directive was expected.
```

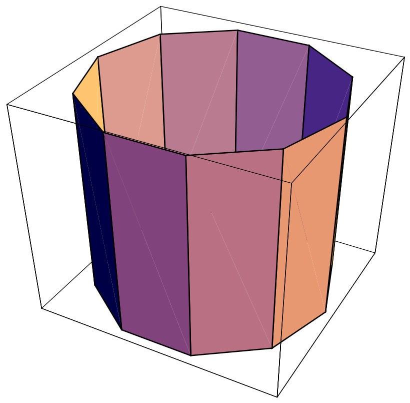
The blue error message is a reminder that Shapes package containing the `Cylinder` graphics primitive was not loaded, so we do it now:

```
In[4]:= << Graphics`Shapes`
        Show[Graphics3D[Cylinder[0.5, 0.5]]]

        Graphics3D::gprim : Cylinder[0.5, 0.5] was encountered
           where a Graphics3D primitive or directive was expected.

Out[5]= - Graphics3D -
```

Typing the command again does not give us the expected graph because we now have two symbols `Cylinder`, and the one introduced before the loading of the package takes precedence. Now the `Remove` command is handy:

```
In[6]:= Remove[Cylinder]
```

```
In[7]:= << Graphics`Shapes`
        Show[Graphics3D[Cylinder[0.5, 0.5]]]; (*  That's better!  *)
```

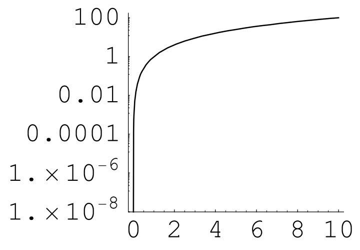


While speaking of packages, you can easily check which packages you have loaded:

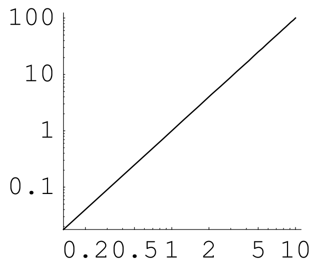
```
In[9]:= $Packages    (*  Check whick packages are loaded  *)
Out[9]= {Graphics`Shapes`, Geometry`Rotations`, Utilities`FilterOptions`,
         Graphics`Common`GraphicsCommon`, Graphics`Graphics`, Global`, System`}
```

Now that the Graphics package is loaded, we can try out the `LogPlot` command:

```
In[10]:= LogPlot[x^2, {x, 0, 10}];
```



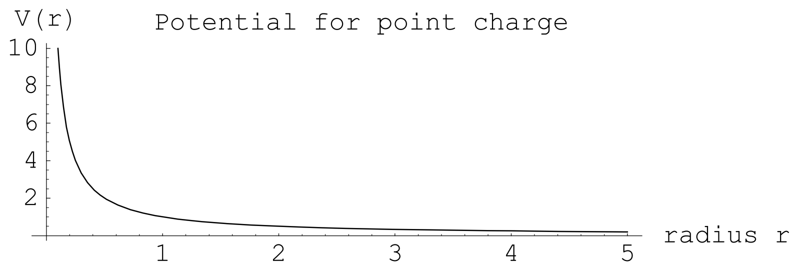
```
In[11]:= (*  Try log-log plot here using LogLogPlot command  *)
         LogLogPlot[x^2, {x, 0, 10}];
```



■ 4.4.2 Labels and Titles (the plot thickens)

Any plot worth looking at is worth explaining. This is done by placing labels along the axes and by placing a title above the curves. Here, try this:

```
In[12]:= Plot[1 / r, {r, 1 / 10, 5}, AxesLabel → {"radius r", "V(r)"},
           PlotLabel → "   Potential for point charge"];
```



Take stock of how we just add a comma after the range list (in braces) and then added the label and title options (with arrows), separated by commas. Look carefully at the syntax for each option, and notice how they are really *rules* with the arrow → pointing towards a list. Since there are two axes, the `AxesLabel` option contains a list of entries, one for the *x* axis, and one for the *y* axis. The `PlotLabel` option requires only one string, hence no curly braces needed. Notice next that the labels and title are enclosed in *double quotes*. When you input a *string* of text to *Mathematica* you must always enclose it in double quotes. The only type of quotes allowed are the double quotes.

The quotes in *Mathematica* are used to define an *alphanumeric string*. A string is just a literal recording of whatever characters are within the quotes; the computer does not try to interpret a string as either variables or numbers. You can put anything you want within the string and it will get pasted into your plot. Modify the previous command so that the words ``abscissa" and ``ordinate" appear in the appropriate places and so that the actual expression being plotted appears in the title:

```
In[13]:=  (*  Plot with your modified labels and title  *)
```

In a later section we will discuss how to place a *legend* on your graphs to identify an individual curve on a multicurve graph. You may have already noticed that we have placed *r* along the "x" axis and *V(r)* along the "y" axis. Since this is a free world, we may also call *y* the independent variable at times. So you see already why the words ordinate and abscissa were invented (to avoid confusion).

4.5 Compound (Abstract) Data Types, *[..]* and *{..}*

As we proceed with our exercises in visualization, you will see how to enter arguments to the `Plot` commands using different types of parentheses. We recognize that some users may prefer just following the rules without questioning them. Nevertheless, the commands will make more sense, and will be easier to generalize, if you have some understanding of the method behind the madness. And so we now take a little excursion in which we define some terms that are frequently used in mathematics and computer science and employed by *Mathematica* commands.

We have already seen a number of ways in which *Mathematica* displays data. Sometimes there's just a single symbol, sometime's there's a bunch of symbols separated by commas, sometimes there's a bunch of things in curly braces, and sometimes not. For example, in Chap. 5 you will see that when the `Solve` command produces several solutions, *Mathematica* separates them with commas and lists them in curly braces:

```
In[14]:= Solve[x^4 - 1 == 0, x]

Out[14]= {{x → -1}, {x → -i}, {x → i}, {x → 1}}
```

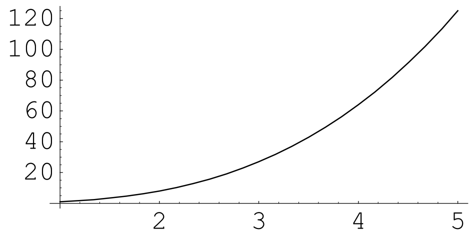
Take note of two types of parentheses here.

Abstract or **Compound Data Types**: An *object* in Computer Science denotes a data type with multiple parts. It may also be called an *abstract* data type, or a *compound* data type. Here the word ``abstract" means that there is more to something than meets the eye, that is, the data type may contain multiple parts. Many of the individual symbols or variables used in *Mathematica* can be replaced by *objects*. We will discuss objects in more depth when we study Java, which is known as an *object oriented* language.

Sequence: A collection of objects separated by commas is called a ***sequence*** in mathematics. Note that the arguments given to this `Solve` command, and indeed to most *Mathematica* commands, are variables or rules separated by commas, and accordingly form sequences. The square brackets indicate an argument to the function (as discussed in Chapter 2), so in these cases the function argument is a sequence.

List: In *Mathematica* the way to make collections of objects is with a ***list***. Lists are important and general structures (objects themselves) that contain comma-separated objects within curly braces {...}. Sometimes the order of objects within a list matters, and sometimes it doesn't. An example of when the order matters can be found in the `Plot` command:

```
In[15]:= Plot[x^3, {x, 1, 5}];
```



Within the second argument list it is clear that order matters because the maximum and minimum are different. However, order does not matter within the list of solutions that the `Solve` command produces. All of the solutions are equally valid. Therefore, it is the purpose of the list which determines whether the order of objects matters or not.

Arrays: Another data type related to vectors and matrices are arrays. We discuss them in Chapter 7.

■ 4.5.1 Several Curves on One Plot

We have seen that the first argument to the `Plot` command is the function to be plotted; e.g:

$$\{x(t),\ v(t),\ a(t)\}$$

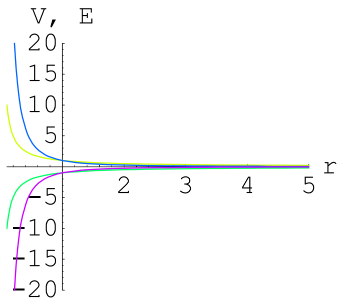
For our electric potential problem, we may want to compare the r dependence of the potential and the magnitude of the electric field due to a positive charge to those due to a negative charge. We know that the potential falls off as $1/r$ and the force as $1/r^2$. In cases such as this, where there are several functions and all are functions of the same independent variable (radius r here), it may be illuminating to plot all functions on the same graph. Since *Mathematica* treats the argument as an object, it can be an abstract data type, and we can substitute the *list*

$$\{1/r,\ -1/r,\ 1/r^2,\ -1/r^2\}$$

as the first argument to the `Plot` command. The fact that we used a list as the object to plot makes sense if you recall that any time a collection of expressions is needed, lists are used in *Mathematica*. In this case the order of the list does not matter, because the order of plotting does not matter. (Also note that for the sake of clarity to you, we try to include spaces after commas. *Mathematica* doesn't care one way or the other.)

Experiment now with the `Plot` command for a list of functions:

```
In[16]:= (*  List as first argument  *)
      Plot[{1 / r, -1 / r, 1 / r^2, -1 / r^2},
        {r, 1 / 10, 5}, PlotRange → {{1 / 10, 5}, {-20, 20}},
        PlotStyle → {{Hue[.2]}, {Hue[.4]}, {Hue[.6]}, {Hue[.8]}},
        AxesLabel → {"r", "V, E"}];
```



Notice that for small *r* the force diverges more rapidly than the potential, while for large *r* the force dies off more rapidly. Finally, note how each function has a different color. This is due to the `PlotStyle` option within the `Plot` command. *Mathematica* does not automatically color the graphs, but rather the user must specify the colors; we will talk about this more later.

■ 4.5.2 Customizing Colors and Line Types

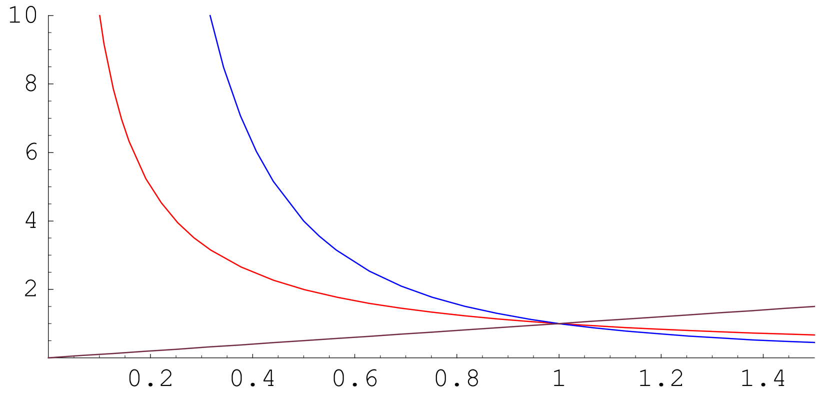
It is important to distinguish the various curves on a plot since *Mathematica* will draw them all with the same size line in the color black unless you specify otherwise. Keep in mind that you want colors and lines that look great on the screen as well as on printouts or projections (green and yellow are often barely visible). Graphics *options* (uses →) are used to specify how the graphical elements should be drawn. The option `PlotStyle` is used to specify the style of a graph:

```
PlotStyle → style            (*  If only one style option is used  *)
PlotStyle → {style1, style2, ...} (*  If several style options are needed,
  ie more than one curve is on the graph  *)
```

Some of the most common styles include those functions which allow graphs to be drawn in color, or grayscale:

- **GrayLevel[x]** with $0 \le x \le 1$, allows for lightening of the image. Values of *x* closer to 1 will make the image appear lighter.
- **Hue[*h*]** is color specification. As *h* is varied from 0 to 1, the color runs from red, yellow, green, cyan, blue, magenta, and back to red.
- **Hue[*h, saturation, brightness*]** specifies colors in terms of their hue, saturation, and brightness. Each has values between 0 and 1.
- **RGBColor[*red, green, blue*]** specifies a color with certain mixture of red, green, and blue components, each between 0 and 1. For example, RGBColor[0,1,0] produces pure green. An easy way to compute an RGB formula is to go to Input menu and choose ColorSelector to click on the color of your choice. The RGB formula for that color will be placed into your *Mathematica* notebook at the cursor position.
- **CMYKColor[*cyan, magenta, yellow, black*]** specifies a color with certain mixture of cyan, magenta, yellow, and black, each between 0 and 1. This is used for printing colored graphics on paper.

```
In[17]:= Plot[{1 / r, 1 / r^2, r}, {r, 0, 1.5},
           PlotRange → {{0, 1.5}, {0, 10}}, PlotStyle → {RGBColor[1, 0, 0],
             RGBColor[0, 0, 1], RGBColor[0.454902, 0.172549, 0.258824]}];
```



In the previous plot the first two curves are colored red and blue respectively, while the ColorSelector was used to pick a maroon color for the third. The `PlotStyle` option is also used to specify the style of line used for various curves. We can add a list of styles containing the type of lines for each graph. An especially effective way to distinguish different curves on the same plot without the use of color, is to draw them with different thickness, which you can try below:

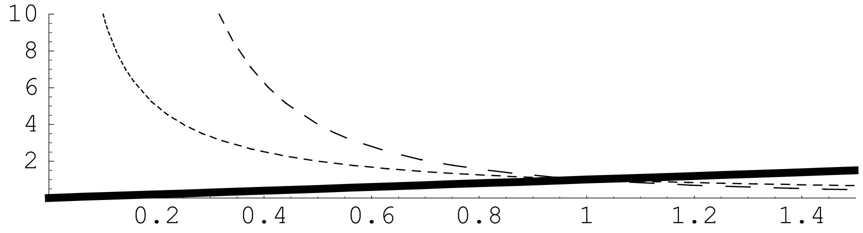
- **Dashing[{*r1, r2,* ...}]** shows the lines as a sequence of dashed segments and spaces of lengths, *r1, r2,* ... repeated cyclically. Each *r* value is given as a fraction of the total width of the graph. If you wish to distinguish curves with dashing, they each need their own `Dashing` function.

- **AbsoluteDashing[{*a1, a2,* ...}]** similar to `Dashing` but uses absolute units to measure dashed segments. The absolute lengths are measured using printer's measure, approximately 1/72 of an inch.

- **Thickness[*r*]** gives the lines a thickness *r* as a fraction of the width of the graph. The default value for 2-D graphs is 0.004.

- **AbsoluteThickness[*r*]** gives the lines a thickness *r* measured in absolute units, defined the same as above.

```
In[18]:= Plot[{1 / r, 1 / r^2, r}, {r, 0, 1.5}, PlotRange → {{0, 1.5}, {0, 10}},
           PlotStyle → {Dashing[{0.01}], Dashing[{0.03}], Thickness[0.01]}];
```



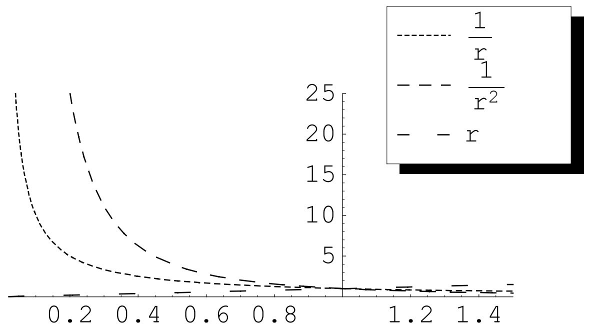
■ 4.5.3 Legends

Legends explain to the reader just what is being plotted with each curve. They are invaluable and do wonders for your presentation. When presenting several curves in one graph, it is important that the viewer not only be able to tell them apart by the different color or line style used for each, but also be given information as to what the different curves represent. It is good practice to explain in the caption below a graph what each curve means, as well as in the text (or in your talk) when the graph gets referenced. However, it is also good practice to have a *legend* in the plot itself explaining what each curve means. Captions and text may get removed, but it is a lot harder to remove a good legend.

The legend option for the `Plot` command is `PlotLegend[{`*`text1, text2, ...`*`}]` which attaches strings *`text1, text2, ...`* to each description in `PlotStyle`, in the order of the curves. If there are more `PlotStyle` descriptions than text descriptions, the text will be repeated cyclically. `PlotLegend` is contained within the `` Graphics`- Legend` `` package and must be loaded before use:

```
In[19]:= << Graphics`Legend`
```

```
In[20]:= Plot[{1 / r, 1 / r^2, r}, {r, 1 / 50, 1.5}, PlotRange → {{1 / 50, 1.5}, {0, 25}},
           PlotStyle → {Dashing[{.01}], Dashing[{.03}], Dashing[{.03, .08}]},
           PlotLegend → {1 / r, 1 / r^2, r}, LegendPosition → {.5, .2}];
```

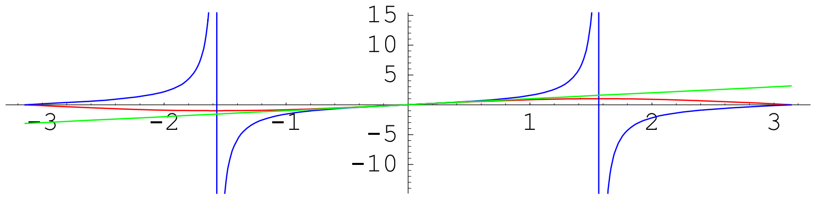


The `LegendPosition` option is used with the legend to place it conveniently. The default position for the legend is in the lower left corner of the graph and will most likely cover important portions of the plots. Hence, you will want to tweak the position coordinates in the command to place the legend where you wish. Other display options for the legend can be found in the ``Graphics`Legend` `` package.

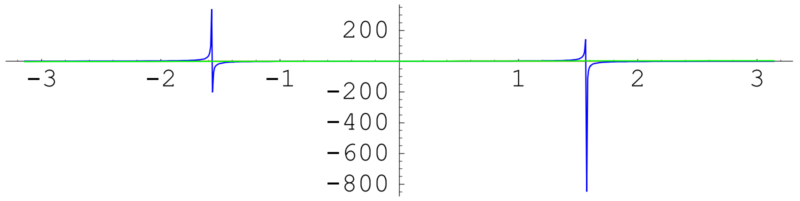
■ 4.5.4 Other Options

There are many ways to customize your graph, and the `Help` browser is a good place to find out about them. Here we look at a few. When *Mathematica* plots, it tries to scale the *x* and *y* axes to show only the most interesting features of the plot. Therefore, if your function gets too large, or has singularities, those parts may be cut off. This is the different from Maple, which automatically adjusts the ordinate's range to accommodate large variations. The `PlotRange` option can be used to specify exact ranges of *x* and *y* for your plot. For example, `Tan[x]` would normally overshadow `Sin[x`, but *Mathematica* automatically limits the *y* range so that all functions are shown:

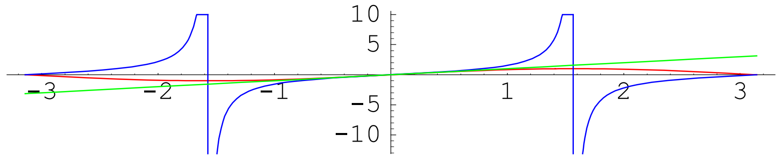
```
In[21]:= Plot[{Sin[x], Tan[x], x}, {x, -Pi, Pi},
          PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 0, 1], RGBColor[0, 1, 0]}];
         (*  Vertical limits automatic  *)
```



```
In[22]:= Plot[{Sin[x], Tan[x], x}, {x, -Pi, Pi}, PlotStyle →
           {RGBColor[1, 0, 0], RGBColor[0, 0, 1], RGBColor[0, 1, 0]}, PlotRange → All];
         (*  PlotRange set to s how all points.  Tan[x] overshadows Sin[x] and x  *)
```



```
In[23]:= Plot[{Sin[x], Min[10, Tan[x]], x}, {x, -Pi, Pi},
          PlotStyle → {RGBColor[1, 0, 0], RGBColor[0, 0, 1], RGBColor[0, 1, 0]}];
         (*  Another way to force limiting the vertical, if need be  *)
```

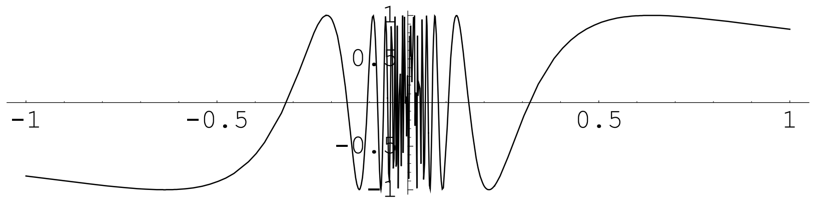


Sometimes a function falls off slowly, and so you might want to see its behavior for values of its argument from negative to positive infinity. While Maple allows plots like this, *Mathematica* appears to be too picky:

```
In[24]:= Plot[Exp[-x^2], {x, -∞, ∞}, PlotLabel -> "Gaussian"];

         Plot::plln :
          Limiting value -∞ in {x, -∞, ∞} is not a machine-size real number.

In[25]:= (*  Try to find a good x range and use
           PlotRange to get an interesting graph of e^(-x^2)  *)
```

Another `Plot` option to explore is `PlotPoints`. *Mathematica* always tries to plot functions as smoothly as possible, and so *Mathematica* will use more points in the region where your function is highly oscillatory, rather the default of 25 points:

```
In[26]:=  Plot[Sin[1 / x], {x, -1, 1}];
```



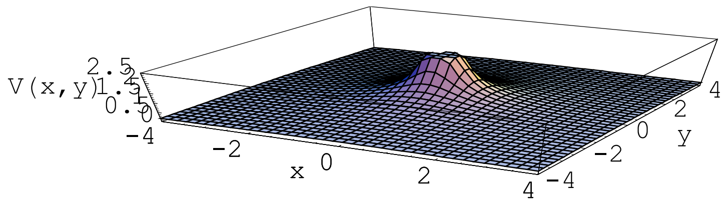
Use `PlotPoints` to sample the above function with more points. Is it better or worse? Remember that the larger the value of `PlotPoints`, the longer it will take *Mathematica* to plot the function, but it can be most helpful at times.

4.6 3D (Surface) Plots of Analytic Functions

We have examined the potential field *V(r) = 1/r* surrounding a single charge as a function of *r*. A 2-D plot is fine for this since there is only one independent variable *r*. However, when the same potential is expresses as a function of the Cartesian coordinate *x* and *y,*

$$V(x,\ y) = \frac{1}{\sqrt{x^2 + y^2}},$$

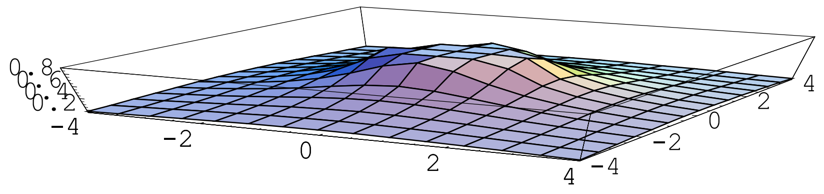
we have two independent variables, *x* and *y*, and so need a 3-D visualization. We get that by creating a world in which the *z* dimension (mountain height) is the value of the potential, and *x* and *y* lie on a flat plane below the mountain. Because the surface we are creating is a 3-D object it is not possible to draw it on a flat screen, and so different techniques are used to give the impression of three dimensions to our brains. We do that by rotating the object, shading it, employing parallax, and so forth.



The command to make a 3-D plot is `Plot3D`, and in many ways is like our old friend `Plot`:

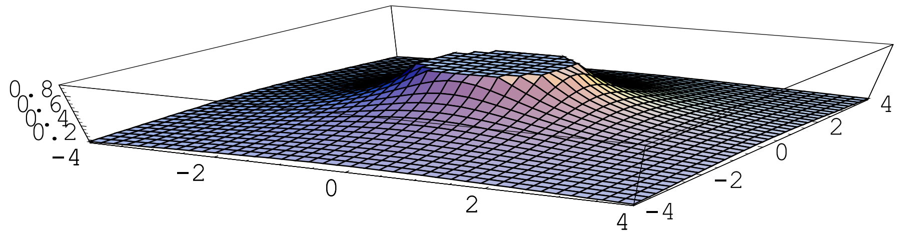
```
In[27]:=  Clear[x, y, V]
          V[x_, y_] := 1 / Sqrt[x^2 + y^2];
          (*  Define function of two variables  *)
          ?V
          Global`V
          V[x_, y_] := 1/√(x²+y²)
```

```
In[30]:= Plot3D[V[x, y], {x, -4, 4}, {y, -4, 4}];        (*  Basic form of Plot3D  *)
```



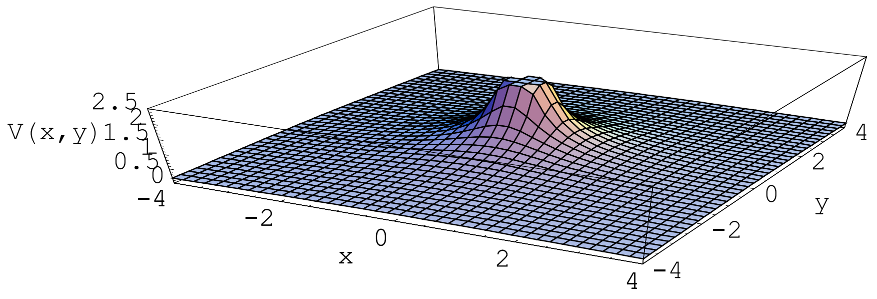
This is a pretty interesting plot showing (what should be) the singular nature of the potential near the origin, but it is rough looking. The option `PlotPoints` specifies the number of points to be used to produce the graph in each direction. The default value for a 3-D plot is 15, but as you can see, this leads to ragged surfaces on many graphs. Let us try the same plot, but with more point:

```
In[31]:= Plot3D[V[x, y], {x, -4, 4}, {y, -4, 4}, PlotPoints → 40];
```



We get more information yet if we extend the *z*-axis range higher. We do that with the `PlotRange` option for the *z* range (and also add some labels):

```
In[32]:= Plot3D[V[x, y], {x, -4, 4}, {y, -4, 4}, PlotPoints → 40,
           PlotRange → {0, 2.5}, AxesLabel → {"x", "y", "V(x,y)"}];
```



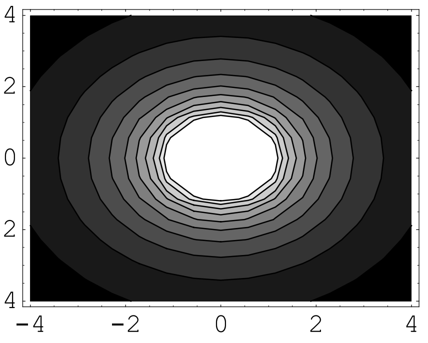
■ 4.6.1 Contours and Equipotential Surfaces

To make your mind understand that different colors mean different potential values, we now use the `ContourPlot` command to produce contour lines showing the different levels of the potential:

```
In[33]:= Clear[x, y, V]
         V[x_, y_] := 1 / Sqrt[x^2 + y^2]; (*  Define a function of two variables  *)
         ?V

         Global`V
```

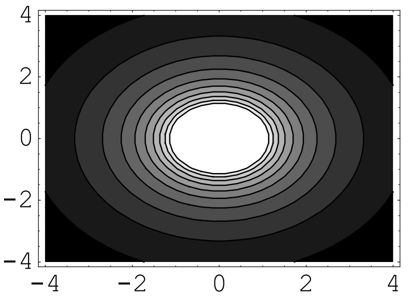
V[x_, y_] := $\frac{1}{\sqrt{x^2+y^2}}$

```
In[35]:= ContourPlot[V[x, y], {x, -4, 4}, {y, -4, 4} ];
```



In analogy to gravity, the contour lines can be thought of as lines of equal elevation. If you walked along a contour line, your elevation would not change. Imagine the plot above in three dimensions: it would look like a mountain with rings around it. Therefore a trip around one of the rings would be at constant elevation. (For a visual cue look at the corresponding Chapter 4.5.1 section in the text book). For our electrical potential problem, the contours correspond to the locust of points in space with the same potential, and are called *equipotential surfaces* (even if they are just lines). Likewise, no work is done if a charge is moved along an equipotential surface (there may be a force on the charge, but it is perpendicular to the surface). We see the circles that are expected for the symmetric case of a single charge, but they are rather grainy. That' s a clue that *Mathematica's* algorithm for contours is probably not evaluating the function with enough points, so we increase the number:

```
In[36]:= ContourPlot[V[x, y], {x, -4, 4}, {y, -4, 4}, PlotPoints → 30];
```



That looks better! However, we prefer the Maple 3-D contour surface in which the contours are on the surface. [*Mathematica* does have a `ContourPlot3D` command under the Graphics package. It plots the surface implicitly defined by *fun[x, y, z] == 0*, and only contour surfaces for specified values. This does not work with our function.]

4.7 Solution: Dipole and Quadrupole Fields

We have used a number of visualization tools to examine the potential field surrounding a single positive and single negative charge. The tools only showed us what we probably knew already, namely that the potential field does not have concave areas in which a charge may remain stably at rest. This was as intended; it is a good idea to learn about new tools on problems for which you know the right answer. We are now in the position to finally investigate the electric potential due to a dipole and tripole. Let us start with the dipole configuration, as shown in Figure 4.2. We know that the potential is

$$V_2(x, y) = \frac{q_1}{\sqrt{(x-1)^2 + y^2}} + \frac{q_2}{\sqrt{(x+1)^2 + y^2}}$$

We start by defining a *Mathematica* function

```
In[37]:= Clear[V, x, y]
         v2[x_, y_, q1_, q2_] := q1 / Sqrt[(x - 1) ^2 + y^2] + q2 / Sqrt[(x + 1) ^2 + y^2];
         ?v2
```

```
Global`v2
```

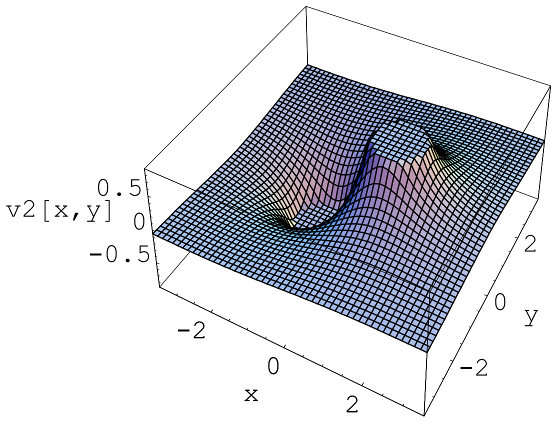
v2[x_, y_, q1_, q2_] := $\frac{q1}{\sqrt{(x-1)^2+y^2}} + \frac{q2}{\sqrt{(x+1)^2+y^2}}$

Now we visualize it with the 3-D plots and contours. First we try a classic dipole with one positive and one negative charge:

```
In[40]:= q1 = 1;
         q2 = -1;
         Print["q1 = ", q1]
         Print["q2 = ", q2]
```

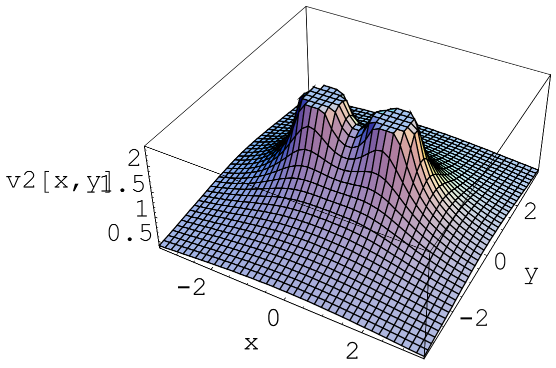
```
q1 = 1
q2 = -1
```

```
In[44]:= Plot3D[v2[x, y, q1, q2], {x, -3.5, 3.5}, {y, -3.5, 3.5},
           AxesLabel → {"x", "y", "v2[x,y]"}, PlotPoints → 50];
```



If you change the viewpoints and replot, you will see that wherever you place a charge it will either roll downhill away from the positive charge, or fall into the hole of the negative charge. This means that there is no stable point. So let us look at two charges of the like sign:

```
In[45]:= q1 = 1;
         q2 = 1;
         Plot3D[v2[x, y, q1, q2], {x, -3.5, 3.5}, {y, -3.5, 3.5},
           AxesLabel → {"x", "y", "v2[x,y]"}, PlotPoints → 40];
         (*  Enter command to make 3-D plot for two positive charges  *)
```



```
In[48]:= (*  Enter command to make 3-D plot for two negative charges  *)
```

Interestingly enough, the figure (which looks like a saddle) has a region between the two peaks where it appears that a charge can remain at rest. In addition, if the charge is displaced along the positive or negative *x* axis, it will roll back towards the midpoint. This means that the midpoint position is stable for disturbances in the *x* direction. However, if the test charge has a component of displacement in the *y* direction, then it will roll away to infinity, clearly not a stable thing to do. This type of shape is known as a *saddle point*. It occurs for two positive or two negative charges. If the charges have unequal values, then the shape gets distorted, but still has the same property. As a check on our analysis, let us look at the contours for this surface:

```
In[49]:= (*  Make a contour 3-D here- still not sure how to do this!!!  *)
```

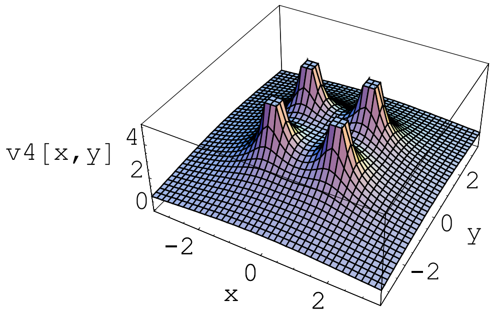
We see the saddlepoint structure as a single point where two equipotential surfaces cross.

Let us now look at the quadrupole potential (we leave the tripole for you). Our intuition tells us that the high degree of symmetry here must lead to a stable position at the center. We define the potential and then we plot it as a 3-D surface and as 3-D contours:

```
In[50]:= v4[x_, y_] := 1 / Sqrt[(x - 1)^2 + (y - 1)^2] + 1 / Sqrt[(x - 1)^2 + (y + 1)^2] +
            1 / Sqrt[(x + 1)^2 + (y - 1)^2] + 1 / Sqrt[(x + 1)^2 + (y + 1)^2];
         ?v4
         Global`v4
```

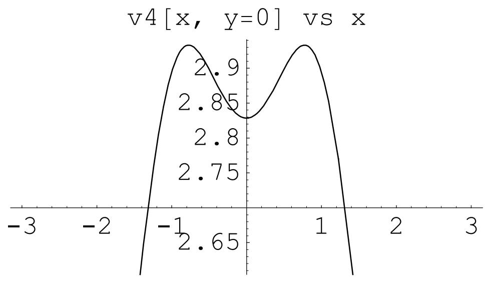
$$v4[x\_, y\_] := \frac{1}{\sqrt{(x-1)^2+(y-1)^2}} + \frac{1}{\sqrt{(x-1)^2+(y+1)^2}} + \frac{1}{\sqrt{(x+1)^2+(y-1)^2}} + \frac{1}{\sqrt{(x+1)^2+(y+1)^2}}$$

```
In[52]:= Plot3D[v4[x, y], {x, -3.5, 3.5}, {y, -3.5, 3.5},
           AxesLabel → {"x", "y", "v4[x,y]"}, PlotPoints → 40];
```

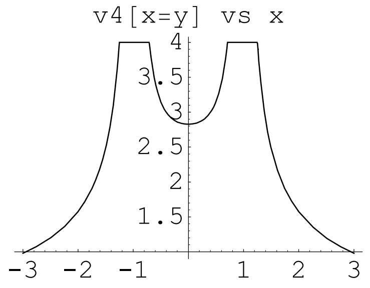


Yes, we do indeed see a large, central flat region surrounded by a lip to hold the charge in. We can check this out further by looking at some slices through the central region:

```
In[53]:= Plot[v4[x, 0], {x, -3, 3}, PlotLabel -> "v4[x, y=0] vs x"];
```



```
In[54]:= Plot[Min[4, v4[x, x]], {x, -3, 3}, PlotLabel -> "v4[x=y] vs x"];
         (*  Use min to cut off hilltops and see details  *)
```



So we see that the central portion of this potential is indeed a flat region with a ``lip" all around it. So a charge placed there will have no force on it (the flatness) and will be stable (the lip).

4.8 Exploration: The Tripole

Repeat the analysis carried out for the dipole and quadrupole now for the tripole. Be sure to make a 3-D plot with labels and title, as well as to include contours. You will also need to slice your plot through its center to verify that you have found a stable point.

4.9 Extension: Yet More Plot Types

■ 4.9.1 2-D Animations

We have just seen that surface-rendering techniques permit us to create images from mathematical functions that give the impression of viewing a true three-dimensional object. This literally gives a new dimension to our visualizations. In addition, if we have plots that show behavior of some quantity as a function of space. If this behavior changes gradually with time, then the observation of a sequence of plots of the spatial dependences, each one for a slightly different time, gives the impression of a continuous evolution of the spatial function in time. The function appears to be alive and, indeed, for this reason, creating a series of snapshots in time is known as *animation*.

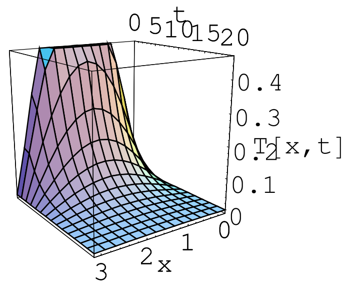
To produce animations we take our familiar 2-D plots and add the dimension of *time* to them. For example, let us say we wanted to show the changing temperature distributions along a metal bar as the bar cooled with increasing time. We could say that we want to show *T1[x]*, *T2[x]*,...*TN[x]*, where the number after *T* is the time. However, it is more elegant and more concise to say that there is only one temperature distribution, and that it is a function of both space and time,

$$T = T[x, t]$$

If the bar was initially hot in the center, and was then cooled over time, the temperature distribution could be approximated by [Krey]

$$T = T(x,\, t) = \sin(x)\, e^{-t} - \frac{1}{9}\sin(3\, x)\, e^{-9\, t}$$

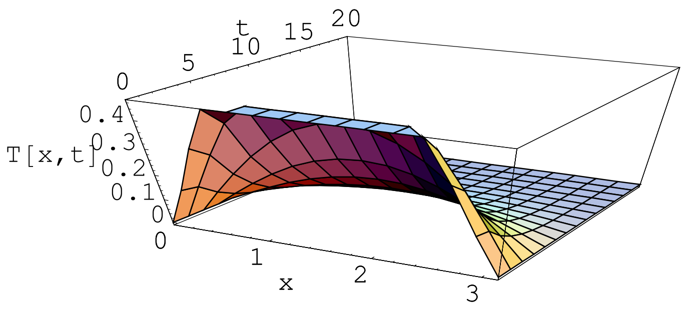
Here is a 3-D surface plot to show the variation of T with both space and time:



In many ways an animation is more natural in displaying the time dependence. In any case, an animation provides another way to visualize a function of two variables, and it is worth looking at to see if it is illuminating.

Make a 3-D surface plot of this temperature distribution from time 0 to 20, and for position 0 to π. Make sure to label the axes so that you know which variable is which. Your result should look like the plot above. Try grabbing this plot and enlarging it, and then rotating it to look like the plot above by using the `3D ViewPoint Selector` and changing the plot command:

```
In[55]:= Plot3D[Sin[x] * Exp[-0.3 * t] - (Sin[3 x] * Exp[-9 * 0.3 t]) / 9, {x, 0, π},
           {t, 0, 20}, BoxRatios → {1, 1, 1}, AxesLabel → {"x", "t", "T[x,t]"}];
```



After we load up the needed package, we will animate this same function by using the `Animate` command:

```
In[56]:= << Graphics`Animation`  (*  Load the needed package  *)
         Names["Graphics`Animation`*"]

Out[57]= {Animate, Animation, AnimationFunction, Frames,
          MovieContourPlot, MovieDensityPlot, MovieParametricPlot, MoviePlot,
          MoviePlot3D, RasterFunction, RotateLights, ShowAnimation,
          SpinDistance, SpinOrigin, SpinRange, SpinShow, SpinTilt}

In[58]:= Clear[x, t]

In[59]:= Animate[ Plot[Sin[x] * Exp[-0.3 * t] - (Sin[3 x] * Exp[-9 * 0.3 t]) / 9, {x, 0, π},
            PlotRange → {0, π}, AxesLabel → {"x", "t"}, Ticks → False], {t, 0, 20} ]
```

Note that at first the result of the command does not look like anything beyond a bunch of static 2-D plots. The first step in animation is the construction of a sequence of images, each slightly different from the other. Next, use your mouse to select the image cell brackets (you must select the bracket containing *all* of the image cells). Then go to the `Cell` menu and choose `Animate Selected Graphics`. *Mathematica* lets you control the direction and speed of the animation with the controls like those on a VCR found in the lower left-hand corner of the screen. Notice the buttons for forward, reverse, stop/start, and continuous loop. We recommend the continuous animation. You can change the speed of the animation with the buttons with up and down arrows. The animation will stop if you click your mouse anywhere else in the notebook.

An animation works by displaying (flipping through) a sequence of slightly modified images. In movie parlance, these images are called *frames*. The more images you have and the less difference between them, the smoother your ``movie" will look. You can include a command option in `Animate` to change the number of frames. The default is 24 frames. A larger number of frames will lead to both a smoother and slower animation.

```
In[60]:= (*  Include a Frames→100 option in the Animate command from above  *)
```

You can also use the `MoviePlot` command to animate functions, and sometimes it is less clumsy than the `Animate` command. A similar series of plots will be created, and you animate them the same way as you did before, by choosing `Animate Selected Graphics`. The `MoviePlot` has the same options as `Plot`, but does not contain the handy `Frames` option found in `Animate`:

```
In[61]:= MoviePlot[Sin[x] * Exp[-0.3 * t] - (Sin[3 x] * Exp[-9 * 0.3 t]) / 9,
          {x, 0, π}, {t, 0, 20}, PlotRange → {0, π}]
```

■ 4.9.2 3-D Animation

Well, if you have been reading and executing up to this point, it is pretty clear what 3-D animations are about. If you have a function of two space coordinates that also varies in time, then you can make a 3-D surface plot to visualize the space dependence at any one time, or an animation to visualize the time dependence. For example, assume the temperature distribution is now the two (space)-dimensional function

$$\left(\mathrm{Sin}[x]\, e^{(-.3\, t)} - \frac{\mathrm{Sin}[3\, x]\, e^{(-9.9\, t)}}{9}\right)$$
$$\left(\mathrm{Sin}[y]\, e^{(-.3\, t)} - \frac{\mathrm{Sin}[3\, y]\, e^{(-9.9\, t)}}{9}\right)$$

This is complicated enough that we will define a *Mathematica* function rather than try to squeeze the long expression into the `MoviePlot3D` command. Other than using the name `MoviePlot3D`, the format of the command is the same as `MoviePlot`. We start with the function T(*x*,*y*,*t*), then give the ranges for each variable, the plot range, and then the labels:

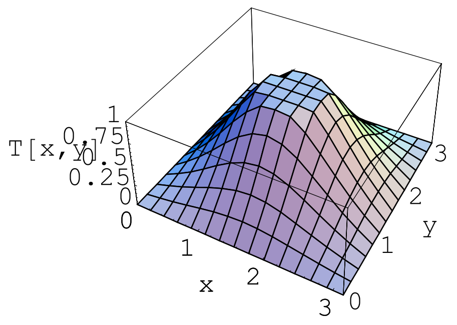
```
In[62]:= << Graphics`Animation`    (*  Load the needed package  *)
```

```
In[63]:= T[x_, y_, t_] := (Sin[x] * Exp[-0.3 * t] - Sin[3 x] * Exp[-9.9 * t] / 9) *
            (Sin[y] * Exp[-0.3 * t] - Sin[3 y] * Exp[-9.9 * t] / 9);
         ?T
```

```
Global`T
```

$$T[x\_, y\_, t\_] := (\mathrm{Sin}[x]\, e^{-0.3\,t} - \tfrac{1}{9}\, \mathrm{Sin}[3\,x]\, e^{-9.9\,t})\, (\mathrm{Sin}[y]\, e^{-0.3\,t} - \tfrac{1}{9}\, \mathrm{Sin}[3\,y]\, e^{-9.9\,t})$$

```
In[65]:= MoviePlot3D[T[x, y, t], {x, 0, π}, {y, 0, π}, {t, 0, 20},
          PlotRange → {0, 1}, AxesLabel → {"x", "y", "T[x,y]"}]
```



■ 4.9.3 Phase Space (Parametric Plots)

In science we often encounter several physical quantities that are simultaneous functions of the same variable. For example, the position *x[t]*, velocity *v[t]*, and acceleration *a[t]* of a mass undergoing simple harmonic motion are all trigonometric functions of time:

$$x(t) = \sin(\omega\, t)$$
$$v(t) = -\omega \cos(\omega\, t)$$
$$a(t) = -\omega^2 \sin(\omega\, t)$$

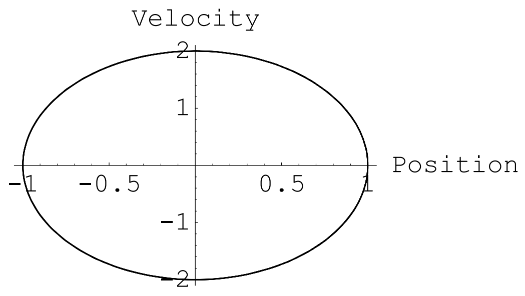
We can easily plot the position and velocity on the same graph:

```
In[66]:= << Graphics`Legend`

In[67]:= Plot[{Sin[ωt], -2 * Cos[ωt]}, {ωt, 0, 8 * π},
            PlotStyle → {{RGBColor[1, 0, 0]}, {RGBColor[0, 1, 0]}},
            PlotLegend → {"x[ωt]", "v[ωt]"}];
```

The graph shows that position and velocity are out of phase, but with the same period. A more direct way to observe the relation of two dependent variables (x and t in our example) as a function of the same independent variable is known as a *phase-space* or *parametric plot*. These types of plots have now proven themselves to be highly illuminating and valuable.

Phase space is an extension of the usual space of position and also includes velocity as if it were a new dimension, along with position. Explicitly, we plot *x(t)* along the abscissa as if it were the independent variable and *y(t)* along the ordinate. In a sense then, a phase-space plot is a plot of the velocity *v(t)* as a function of position *x(t)*, that is, a plot of *v(x)*. In general, there might be some complicated mathematics needed to analytically eliminate the time dependences of these two functions so that they can be expressed in terms of each other. However, it is pretty easy to do this graphically (numerically), and that is what *Mathematica* does. Explicitly, *Mathematica* just breaks up the total time interval *T* into a number of steps, and then records the pair of values *(x,v)* for each time step. These values then get plotted as *v(t) versus x(t)* with the *phase-space plot* `ParametricPlot`:

```
In[68]:= ParametricPlot[{Sin[ωt], -2 * Cos[ωt]},
           {ωt, 0, 8 * π}, AxesLabel → {"Position", "Velocity"}];
```



Note that the syntax of the command is very similar to that of our old friend `Plot`. A list of two functions is used as the function argument, and the range of time values is listed separately. The general syntax for a 2-D parametric plot is

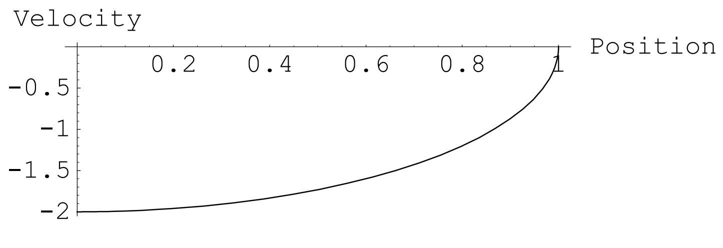
```
ParametricPlot[{x[t], y[t]}, {t, a, b}, options]
```

where *t* is known as the parametric variable, or simply parameter, and *x[t]* and *y[t]* denote the horizontal and vertical functions, respectively.

As far as the output goes, this phase-space plot looks like an ellipse. Yet note that it has properties that agree with the observations we have made before: when the mass is at its maximum positions, at the extreme right and left edges of the ellipse, the velocity is zero. When the mass has it maximum speed, at the top and bottom of the ellipse, it has zero position, that is, it is passing through its equilibrium position. So while a harmonic oscillator can be described via a complicated set of position, velocity, and acceleration functions, in phase space its motion is an elliptical orbit on which the mass passes over and over. This geometric approach has proven to be a simpler way to understand oscillatory motion. Indeed, the general value of phase-space plots is that they convert dynamical relations into geometric ones that are easier to visualize (the point of this chapter after all). For our simple oscillator, a sine relation along one axis and a cosine relation along the other converts into a simple ellipse.

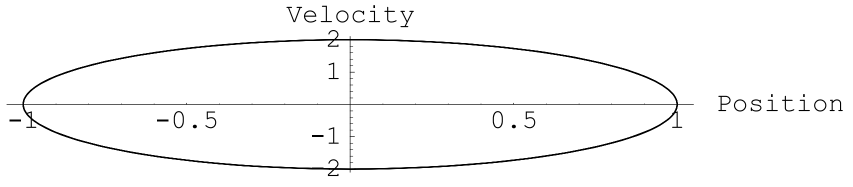
You may be wondering in looking at this graph, just how we hit upon the exact range of *ωt* values for which the graph exactly closes on itself. Well, we really did not. In fact, our plot covers two full cycles and plots them on top of each other (which you cannot see). If, on the other hand, your phase-space plot did not form a closed figure, then you would need to run for more values of the time. Try out smaller and smaller ranges for the phase *ωt* in the plot command until you can generate 1, 1/2, and 1/4 of an ellipse:

```
In[69]:= ParametricPlot[{Sin[ωt], -2 * Cos[ωt]},
           {ωt, 0, .5 * π}, AxesLabel → {"Position", "Velocity"}];
```



As you can see from looking at the monitor in front of you, visual displays tend to be broader than they are high. Accordingly, graphs tend to get stretched horizontally (``scaled") in order to fill the screen. While this is not normally a concern if the graph looks good, it is if you are trying to determine the actual shape of a geometrical figure. *Mathematica* has a `AspectRatio` option designed to determine the height-to-width ratio of a graph. The default value is set to 1/`GoldenRatio`, where `GoldenRatio` is $(1 + \sqrt{5})/2$. This is known to be an eye-pleasing ratio. However, it can be changed to scale both axes identically by setting `AspectRatio-> Automatic`, or any ratio of vertical to horizontal axis length with `AspectRatio->`*`ratio`*. Use the execution group below to try several ratio values and reshape this ellipse back into a circle:

```
In[70]:= ParametricPlot[{Sin[ωt], -2*Cos[ωt]}, {ωt, 0, 8*π},
          AxesLabel→{"Position", "Velocity"}, AspectRatio→.75];
         (*  Play with the AspectRatio option  *)
```



■ 4.9.4 Vector Fields: `PlotVectorField`

Consider again the electric dipole shown in Figure 4.1. The problem we examined dealt with the electric potential for this type of system. While potentials are easier to compute and visualize than fields, it is usually fields that are related to forces and thus measured in experiments. As an extension of our previous work, we now visualize the electric field *E(x,y)* for the dipole. Mathematically, we can determine the electric field as the derivative of the potential using the techniques of vector calculus. This is complicated because the electric field is a vector field with components in the *x*, *y*, and *z* directions. While we can visualize each component of a vector field individually, generally it is more illuminating to examine magnitude and direction of the field. Let us see how *Mathematica* does that.

For the dipole in Figure 4.1, the electric potential has the vector form

$$E = \frac{q_1(r-r_1)}{(|r-r_1|)^2} + \frac{q_2(r-r_2)}{(|r-r_2|)^2}$$

where the *E* and *r*'s in this equation are all vector quantities. For unit charges located a distance 1 and -1 along the *x* axis, the *x* and *y* components of *E* are:

```
In[71]:= Ex[x_, y_] := (x+1) / ((x+1)^2 +y^2) - (x-1) / ((x-1)^2 + y^2);
         ?Ex
         Global`Ex
```

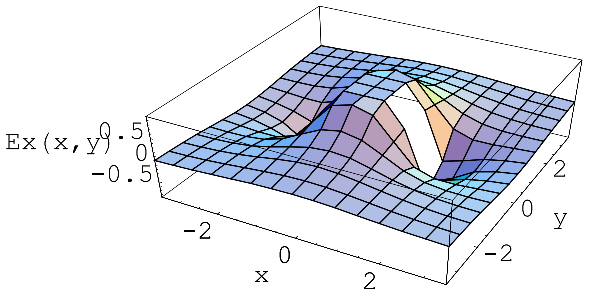
Ex[x_, y_] := $\frac{x+1}{(x+1)^2+y^2} - \frac{x-1}{(x-1)^2+y^2}$

```
In[73]:= Ey[x_, y_] := y / ((x+1)^2 +y^2) - y / ((x-1)^2 + y^2);
         ?Ey
         Global`Ey
```

Ey[x_, y_] := $\frac{y}{(x+1)^2+y^2} - \frac{y}{(x-1)^2+y^2}$

We can visualize these individual components in two surface plots:

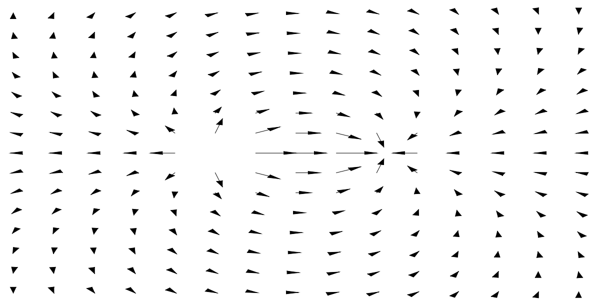
```
In[75]:= Plot3D[Ex[x, y], {x, -3.5, 3.5},
           {y, -3.5, 3.5}, AxesLabel → {"x", "y", "Ex(x,y)"}];
```



While this does show the force components at each point in space, it is rather hard to get a good feel for the magnitude and direction of the force acting on a test charge at each point in space. This is obtained with the `PlotVectorField` command, and hence we need to load the `PlotField.m` package:

```
In[76]:= << Graphics`PlotField`
```

```
In[77]:= PlotVectorField[{Ex[x, y], Ey[x, y]}, {x, -3.5, 3.5}, {y, -3.5, 3.5}];
```



We see here the direction of the field given by the direction of the arrows, and the magnitude given by the length.

Although not often seen in elementary classes, the real world is actually three dimensional. Accordingly, the electric field actually has three components:

```
In[78]:= Clear[Ex, Ey]
         Ex[x_, y_, z_] :=
           (x + 1) / ((x + 1) ^2 + y^2 + z^2) - (x - 1) / ((x - 1) ^2 + y^2 + z^2);
         ? Ex

         Global`Ex
```

Ex[x_, y_, z_] := $\frac{x+1}{(x+1)^2+y^2+z^2} - \frac{x-1}{(x-1)^2+y^2+z^2}$

```
In[81]:= Ey[x_, y_, z_] := y / ((x + 1) ^2 + y^2 + z^2) - y / ((x - 1) ^2 + y^2 + z^2);
         ? Ey

         Global`Ey
```

Ey[x_, y_, z_] := $\frac{y}{(x+1)^2+y^2+z^2} - \frac{y}{(x-1)^2+y^2+z^2}$

```
In[408]:= Clear[B, M, m, matrix, v, f]
          v = {vx, vy, vz}                    (*  Create v  *)
          Print["v = ", MatrixForm[v]]
          (*  As column vector  *)

Out[409]= {vx, vy, vz}
```

$$v = \begin{pmatrix} vx \\ vy \\ vz \end{pmatrix}$$

```
In[411]:= v[[1]]                    (*  First element of vector  *)

Out[411]= vx
```

Since *Mathematica* uses lists to represent vectors, there is no distinction as to whether the vector is a row or column vector. While this is a nice feature, if desired, we can create a row vector as a matrix with one row, *i.e.*, a *1 x N* matrix. Likewise, a column vector is the same as an *N x 1* matrix. Remember, since explicit input of matrix elements are done row-by-row, a column matrix requires a list of lists, with each sublist containing the item for that row:

```
In[412]:= Ucol = {{10}, {20}, {30}}; (*  Explicit column vector via list of lists  *)
          Print["Ucol = ", MatrixForm[Ucol]]
```

$$Ucol = \begin{pmatrix} 10 \\ 20 \\ 30 \end{pmatrix}$$

```
In[413]:= Ucol[[2, 1]]              (*  Look at element in matrix  *)

Out[413]= 20

In[414]:= Urow = {{a, b, c}};   (*  Explicit row vector via list of lists *)
          Print["Urow = ", MatrixForm[Urow]]

          Urow = ( 0.015  2  9 )

In[416]:= Urow[[1, 3]]  (*  Look at individual element, OK because matrix  *)

Out[416]= 9

In[417]:= Urow[[3]]    (*  Look at vector, not OK because is matrix  *)

          Part::partw : Part 3 of {{0.015, 2, 9}} does not exist.

Out[417]= {{0.015, 2, 9}}⟦3⟧
```

■ 7.4.3 Arrays

Arrays in *Mathematica* are essentially lists. They can be multi-dimensional and are then considered lists of lists, an example of which is a matrix. Arrays contain sequences of expressions, each specified by a certain index. Each expression is an element of the array and usually given by a function. The `Array[f,n]` command generates a list of length *n* with specified elements, *f[i]*, or nested lists with elements $f[i_1,\ i_2,\ \ldots]$. You access the elements as you would from a list with the `[[ ]]` notation:

```
In[418]:= Clear[f]
```

```
In[83]:= Ez[x_, y_, z_] := z / ((x + 1) ^2 + y^2 + z^2) - z / ((x - 1) ^2 + y^2 + z^2);
         ?Ez
```

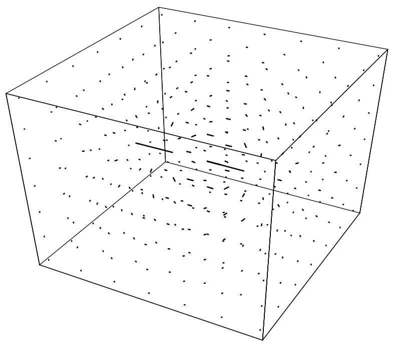
```
Global`Ez
```

$$\text{Ez[x\_, y\_, z\_] :=} \frac{z}{(x+1)^2+y^2+z^2} - \frac{z}{(x-1)^2+y^2+z^2}$$

We visualize a 3-D vector field with the `PlotVectorField3D` command found in the `PlotField3D.m` package:

```
In[85]:= << Graphics`PlotField3D`
```

```
In[86]:= PlotVectorField3D[{Ex[x, y, z], Ey[x, y, z], Ez[x, y, z]},
           {x, -3.5, 3.5}, {y, -3.5, 3.5}, {z, -3.5, 3.5}];
```



Note that the output from the `PlotVectorField3D` command is a 3-D plot that can be rotated using the `3D View Point Selector` under the `Input` menu, having the usual options.

■ 4.9.5 Energy Conservation and Implicit Plots

In a preceding section we looked at the position *x[t]* and velocity *v[t]* of an oscillator, each as a function of time, and showed how a parametric plot can be made from them. *Mathematica* solves numerically for the function *x[v]* or *v[x]*. There may also be cases where you know some functional relation between two variables, say *x* and *v*, and wish to make a plot of *x* versus *v*. For example, let us say that we have a spring with a nonlinear force law so that the potential energy stored in it is

$$V(x) = k x^6$$

The kinetic energy of a mass attached to this spring is, as always,

$$K = \frac{m v^2}{2}$$

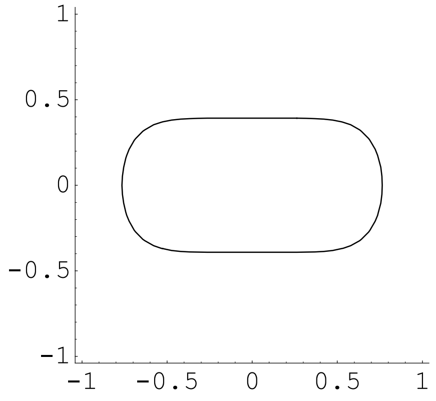
Since we know that the sum of kinetic plus potential energy is conserved, this means we have an implicit relation between position and velocity, namely,

$$E = V + K$$
$$E = k x^6 + \frac{m v^2}{2}$$

This last equation permits us to use *Mathematica's* `ImplicitPlot` command to plot *x versus v* if we have explicit values for the constants:

```
In[87]:= << Graphics`ImplicitPlot`  (*  Load the needed package  *)
```

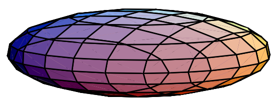
```
In[88]:= ImplicitPlot[5 * x^6 + (13 / 2) * v^2 == 1, {x, -1, 1}, {v, -1, 1}];
         (*  Create an implicit plot of x versus v  *)
```



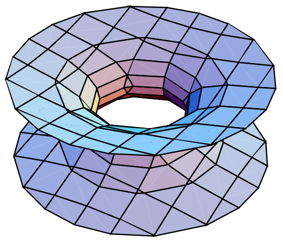
```
In[89]:= (*  See how this phase space diagram changes
           if the potential energy varies as the 5 th power of x  *)
```

In analogy to the 2-D implicit plots we just made for a nonlinear oscillator, the function `ContourPlot3D` plots the 3-D surface defined by some implicit relation between *x*, *y*, and *z*:

```
In[90]:= Clear[x, y, v] (*  Clear existing variables and load package  *)
         << Graphics`ContourPlot3D`

In[92]:= ContourPlot3D[x^2 + y^2 + z^2 / 3 - 1,
           {x, -1, 1}, {y, -1, 1}, {z, -2, 2}, Boxed → False];
```



```
In[93]:= ContourPlot3D[Exp[z^2] - Sqrt[x^2 + y^2],
           {x, -3, 3}, {y, -3, 3}, {z, -1, 1}, Boxed → False];
```



■ 4.9.6 Polar Plots

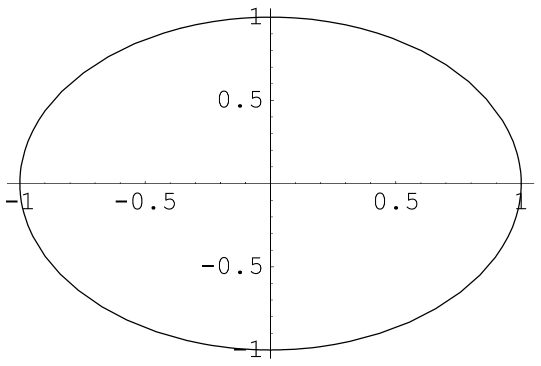
A polar plot is the representation *r(θ)* of a function that is created by placing a point a distance *r(θ)* from the origin for different values of *θ*. If *r* were independent of *θ*, then the polar plot would be a circle. Other dependencies are less obvious until you see them. Polar plots can be highly illuminating since the angle on the plot corresponds to the actual angle of the function's argument, and so the shape of the plot lets you visualize the variation of the function in actual space.

Making polar plots is done using the `PolarPlot` command, where *θ* is the independent variable and *r = f[θ]* as *θ* varies from *θmin* to *θmax*:

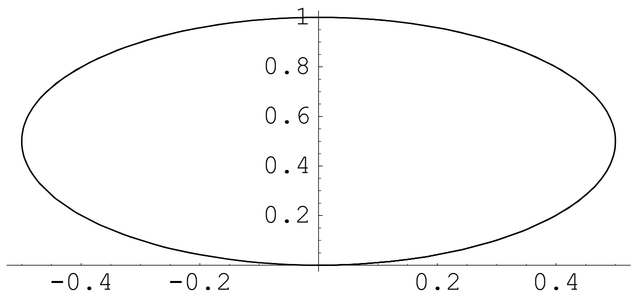
```
PolarPlot[f[θ], {θ, θmin, θmax}]
```

To get a feel for how this works, let us make a polar plot of a function that is independent of *θ*, and then one of the familiar Sin[*θ*]. The `PolarPlot` command is found in the `Graphics.m` package, so we will load it again to make sure we have it:

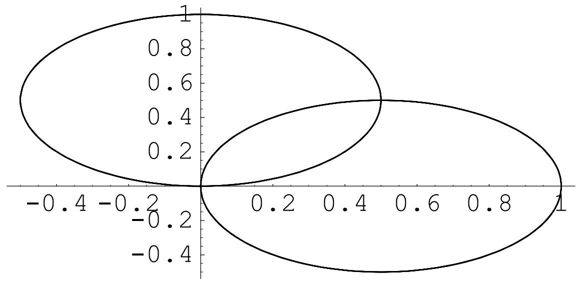
```
In[94]:= << Graphics`Graphics`

In[95]:= r = 1;
         PolarPlot[r, {θ, 0, 2π}];
```



```
In[97]:= PolarPlot[Sin[θ], {θ, 0, 2*π}];
```



```
In[98]:= PolarPlot[{Cos[t], Sin[t]}, {t, 0, 4π}];
         (*  Several polar graphs on one set of axes  *)
```

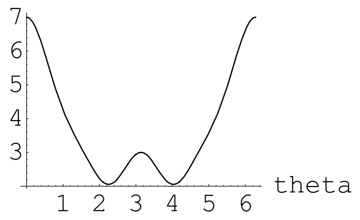


As a more realistic example, consider the expression for the intensity of low-energy X-rays scattered off a reflecting sphere as a function of the scattering angle, or radiation pattern around antenna:

$$\sigma[\theta] = 3 + 2\,\mathrm{Cos}[\theta]^4 + 2\,\mathrm{Cos}[\theta]$$

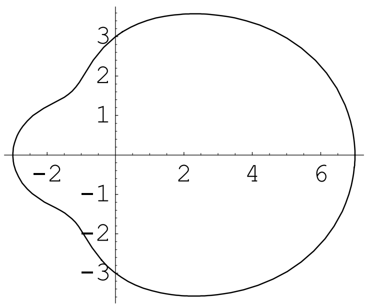
Visualize this function with a 2-D plot of $\sigma[\theta]$ *versus* scattering angle. You should get a plot like the one below:

```
In[99]:= (*  Use Plot[...] here  *)
```



It may not be obvious that this graph means that there is a strong peak in the forward direction along the beam at $\theta = 0$, and that there is a weaker amount of scattering back into the beam at $\theta = \pi$. However, if we plot this function in a polar plot these features become evident:

```
In[100]:= PolarPlot[3 + 2 Cos[θ]^4 + 2 Cos[θ], {θ, 0, 2 π}];
```



Notice how you can intuitively feel where the scattering is large and where it is small. Notice too, that the form of the `PolarPlot` command is very similar to that of the `Plot` command.

■ 4.9.7 Surface Plots of Complex Functions*

At present we have not found a ComplexPlot command or similar 3-D version of complex plotting in version 4.100 of *Mathematica*. However, the corresponding Maple section on this topic is useful.

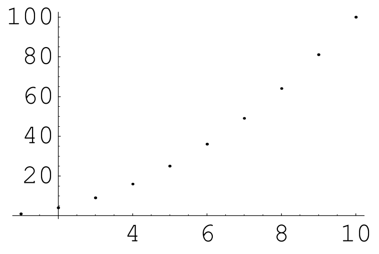
■ 4.9.8 Plotting Lists with `ListPlot`

Say we want to plot a set of data points of the form $x_1, y_1, x_2, y_2, x_3, y_3, \dots x_N, y_N$. To do that we generate a list of numbers with the `Table` command:

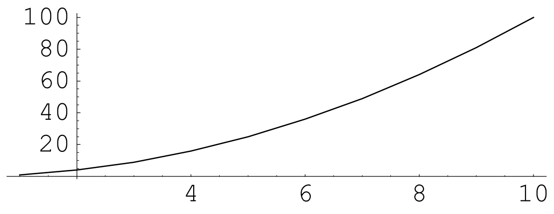
```
In[101]:= mylist = Table[i^2, {i, 10}]

Out[101]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

The order is preserved since this is a list. To make the plot we use the `ListPlot` command. This command creates two-dimensional plots of points from either a list of *y* values or a list of ordered pairs of points, *{x,y}*. We have the first case, and the *x* values are taken to be 1, 2, ..., whereas in the second case, you can specify the *x* values. The option `Plot-Joined -> True` connects the points by line segments, and this should yield a single-valued function if the points are sequentially ordered. Let us try both below `PlotJoined` options below:

```
In[102]:= ListPlot[mylist];
```



```
In[103]:= ListPlot[mylist, PlotJoined → True];
```



■ 4.9.9 Creating Simple Figures: `Graphics` and `Show`

Here we give some examples of the use of the `Graphics` command using graphics primitives `Point` and `Line` along with the `Show` command to create simple figures of a barbell (we use the figure in Chapter 7). There are three steps involved. First we create the shapes with the lines using the `Line` graphics primitive, and then create the data points at the corners using the `Point` graphics primitive. Finally we use *Mathematica's* `Graphics` and `Show` commands to display the lines and points on the same graph. We do this first for a 2-D plot with `Graphics`, and then for a 3-D plot with `Graphics3D`. The 3-D plot is particularly useful as it permits the viewer to rotate the figure to gain different perspectives.

In 2-D we give each of the points as a list of doublets $\{x, y\}$:

```
In[104]:= p1 = Point[{-0.7, 0.7}]
          p2 = Point[{0.7, -0.7}]

Out[104]= Point[{-0.7, 0.7}]

Out[105]= Point[{0.7, -0.7}]
```

The `Line` command is a graphics primitive that represents a line joining a sequence of points. Here we would like a line to join our points and need to enter in their coordinates as a nested list:

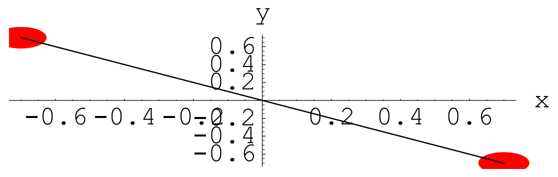
```
In[106]:= ln1 = Line[{{-0.7, 0.7}, {0.7, -0.7}}]

Out[106]= Line[{{-0.7, 0.7}, {0.7, -0.7}}]
```

Since both `Point` and `Line` are graphics primitives, we use the `Graphics` command to represent the graphical image of each, and then use the `Show` command to display the graphical images.

`Graphics[`*`primitives, options`*`]`

The shape, size, and color, of the primitives are contained in a list for the first argument to the `Graphics` command, while the options are the second argument. For example, the axes and plot information are set by the options. However, if you are displaying more than one graphic on the same graph with `Show`, the axes information should be specified in the first graphic, and the graphics are separated by commas:

```
In[107]:= Show[Graphics[{RGBColor[1, 0, 0], PointSize[0.1], {p1, p2}},
             Axes → True, AxesLabel → {"x", "y"}], Graphics[ln1]];
```

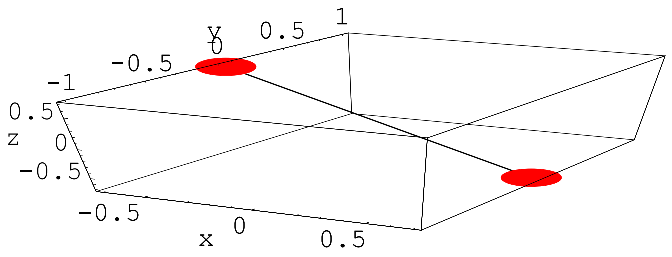


In 3-D we give the points as a list of triplets $\{x, y, z\}$, and the line will now connect those points:

```
In[108]:= p1 = Point[{-0.7, 0, 0.7}]
          p2 = Point[{0.7, 0, -0.7}]
          ln1 = Line[{{-0.7, 0, 0.7}, {0.7, 0, -0.7}}]

Out[108]= Point[{-0.7, 0, 0.7}]

Out[109]= Point[{0.7, 0, -0.7}]

Out[110]= Line[{{-0.7, 0, 0.7}, {0.7, 0, -0.7}}]
```

Now we will use the `Graphics3D` command for the graphics objects. The same `Show` command can be used to display 2-D and 3-D graphics:

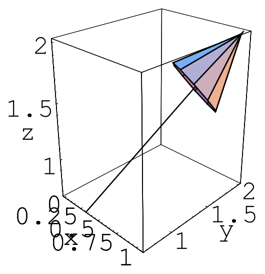
```
In[111]:= Show[Graphics3D[{RGBColor[1, 0, 0], PointSize[0.1], {p1, p2}},
             Axes → True, AxesLabel → {"x", "y", "z"}], Graphics3D[ln1]];
```



■ 4.9.10 Plotting Vectors: `Arrow`*

Mathematica's ability to do linear algebra is discussed in Chapter 7. We can define vectors, matrices, and arrays of arbitrary sizes and dimensions. *Mathematica* provides some easy-to-use tools for visualizing vectors and matrices, in particular `Arrow`, `Arrow3D`, and `ListPlot`.

In[112]:=



Visualization of Vectors with `Arrow3D`

The `Arrow` command creates a 2-D graphics primitive from points specifying the base and tip of the vector. Likewise, `Arrow3D` creates a 3-D graphics primitive from two points in 3-D. [You may have to copy `Arrow3D`.m into the `Graphics` folder under your installation of *Mathematica*'s `StandardPackages`.] The `Show` command lets you place several `Graphics` or `Graphics3D` arrows together.

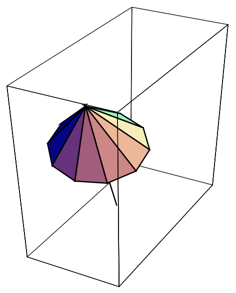
3-D Arrows

```
In[112]:= << Graphics`Arrow3D`  (*  Load Arrow3D package  *)

In[113]:= Omega = {1, -3, 6};    (*  Define vector Omega  *)
          Print["Omega = ", Omega]

          Omega = {1, -3, 6}
```

```
In[114]:= L = {6, 0, 6};          (*  Define vector L  *)
          Print["L = ", L]

          L = {6, 0, 6}
```

```
In[115]:= w = Arrow3D[{0, 0, 0}, Omega]; (*  Assign object w to arrow of  Omega  *)
```

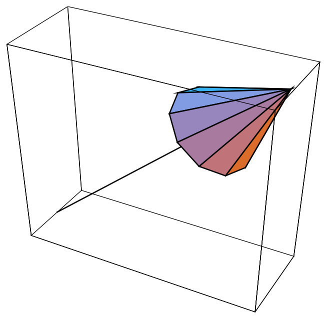
```
In[116]:= Show[Graphics3D[w]];
```



Go up to the `Input -> 3D ViewPoint Selector` and change the `Graphics3D` command above to rotate the omega arrow.

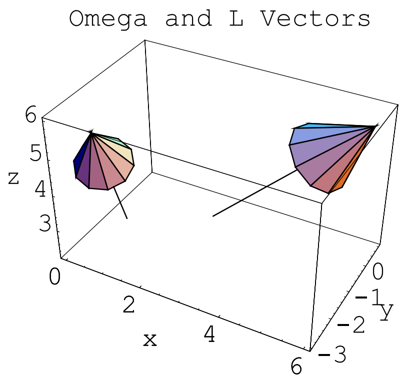
```
In[117]:= l = Arrow3D[{0, 0, 0}, L]; (*  Assign object l to arrow of L  *)
```

```
In[118]:= Show[Graphics3D[l]]; (*  Rotate this arrow as well  *)
```



We now have visualizations of both L and Ω that we can rotate. Note that we defined the arrows to start at the origin and end at the defining vector location. We place them on the same graph by placing both of the `Graphics3D` commands separated by commas in `Show`:

```
In[119]:= Show[Graphics3D[w, Axes → True, AxesLabel → {"x", "y", "z"},
            PlotLabel -> "Omega and L Vectors"], Graphics3D[l]];
```

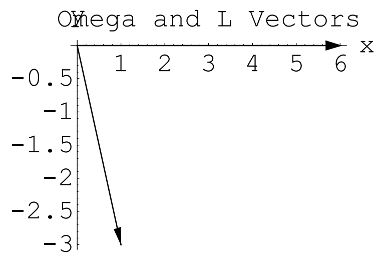


Since the `Arrow3D` is a fairly crude primitive, changing the color of the arrows is not an option (believe me, I have tried). Also, both of the arrows are defined to start at the origin, however the tails do not appear to meet at the origin as expected.

2-D Arrows

```
In[120]:= << Graphics`Arrow`       (*  Load Arrow package  *)

In[121]:= Clear[Omega, w, L, l]
          Omega = {1, -3};
          L = {6, 0};
          Print["L = ", L]
          Print["Omega =", Omega]

          L = {6, 0}

          Omega ={1, -3}

In[126]:= w = Arrow[{0, 0}, Omega]; (*  Assign object w to arrow of Omega  *)

In[127]:= l = Arrow[{0, 0}, L];  (*  Assign object l to arrow of L  *)
```

```
In[128]:= Show[Graphics[w, Axes → True, AxesLabel → {"x", "y"},
             PlotLabel → "Omega and L Vectors"], Graphics[{l, RGBColor[1, 0, 0]}]];
           (*  Display Omega and L arrows on one graph  *)
```



Notice that when we create the graphics objects for the arrows, we tried to give an option to change the color of the arrow, but we are unable to. There are limited options for the `Arrow` and `Arrow3D` graphics primitive. However, we are still able to give the usual options for the plots within the `Graphics` and `Graphics3D` commands. So it is by the use of the `Graphics` command that we are able to place labels and titles, as well as control the scaling to make the vertical and horizontal sizes true. It is by the `Show` command that we are able to place more than one graphics object on a single graph and display it.

4.10 Visualizing Numerical Data

■ 4.10.1 2-D Plots of Data

Most realistic computations in science produce numerical output, not analytic functions. While there is more work involved in plotting numerical data than there is for an analytic function, *Mathematica* is up to the task and actually has a number of ways to make 2-D plots of numerical data. Here we demonstrate the use of `ListPlot` from the standard `Graphics` package and `Histogram` from the `DescriptiveStatistics` package.

■ 4.10.2, 3 Numerical Plots: `ListPlot (scatterplot)`

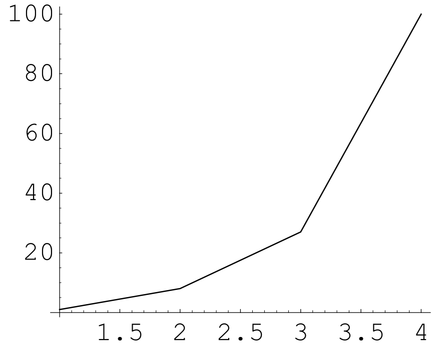
The `ListPlot` command creates a 2-dimensional plot from a list of numerical data values, as we have already seen. If only the *y* values of the data are given, say as the four-element list

```
Ydata = {1, 8, 27, 100},
```

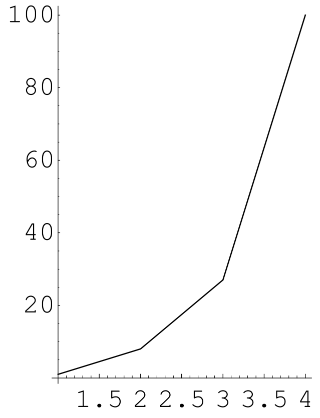
then `ListPlot` will assign *x* values as 1, 2, 3, 4, that is, the data points are

```
(1, 1), (2, 8), (3, 27), (4, 100)
```

```
In[129]:= ListPlot[{1, 8, 27, 100}, PlotJoined → True];
          (*  Plot these y values with x as order number and connect points  *)
```



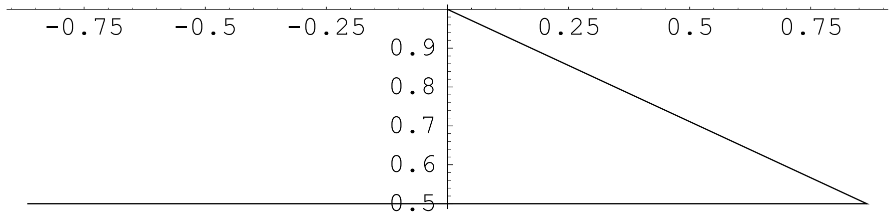
```
In[130]:= Ydata = {1, 8, 27, 100} (*  Enter y data into list Ydata  *)

Out[130]= {1, 8, 27, 100}

In[131]:= ListPlot[Ydata, PlotJoined → True];
          (*  Plot data object Ydata with x as order number  *)
```



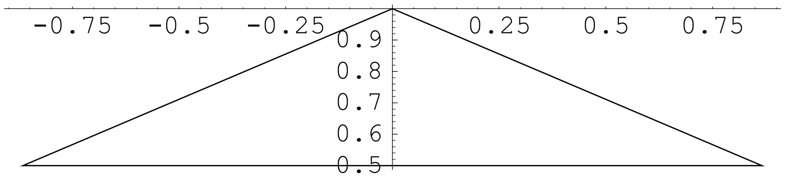
The default value of the `PlotJoined` option is `False`, so in order to connect the points in the order in which they are plotted, you need to set this option to `True`. If you want to give explicit *x* values to your data, then you can place each (x_i, y_i) coordinate value in its own 2-element list, and make a big list of these small lists:

```
{{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}}
```

```
In[132]:= ListPlot[{{0, 1}, {Sqrt[3] / 2, 1 / 2}, {-Sqrt[3] / 2, 1 / 2}},
           PlotJoined → True];
          (*  Plot (xᵢ, yᵢ) values in a list and connect points  *)
```



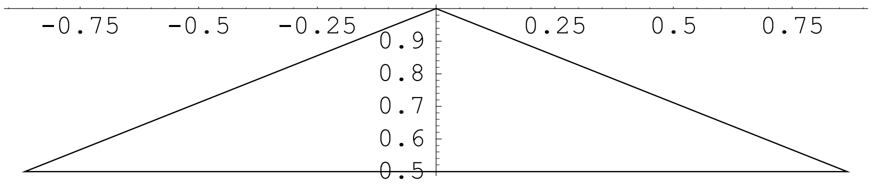
```
In[133]:= ListPlot[{{0, 1}, {Sqrt[3] / 2, 1 / 2}, {-Sqrt[3] / 2, 1 / 2}, {0, 1}},
           PlotJoined → True]; (*  Repeat first point to make a figure  *)
```



```
In[134]:= XYdata = {{0, 1}, {Sqrt[3] / 2, 1 / 2}, {-Sqrt[3] / 2, 1 / 2}, {0, 1}}
           (*  Enter (x,y) data into list XYdata  *)

          General::spell1 : Possible spelling error: new
             symbol name "XYdata" is similar to existing symbol "Ydata".
```

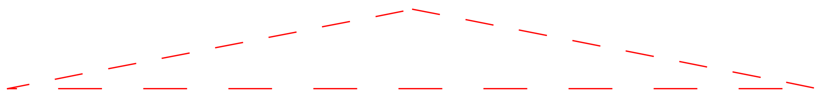
Out[134]= $\{\{0, 1\}, \{\frac{\sqrt{3}}{2}, \frac{1}{2}\}, \{-\frac{\sqrt{3}}{2}, \frac{1}{2}\}, \{0, 1\}\}$

```
In[135]:= ListPlot[XYdata, PlotJoined → True]; (*  Plot XYdata  *)
```

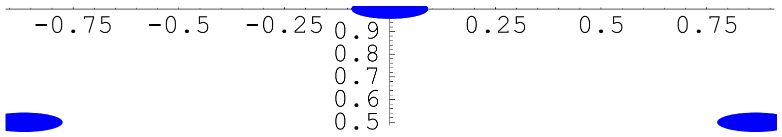


These are the basic commands. The `PlotStyle` option can be used to change the style of points or line. You can control the size and color of points, as well as and the thickness, color, and style of lines used to connect the points if `PlotJoined` is set to `True`:

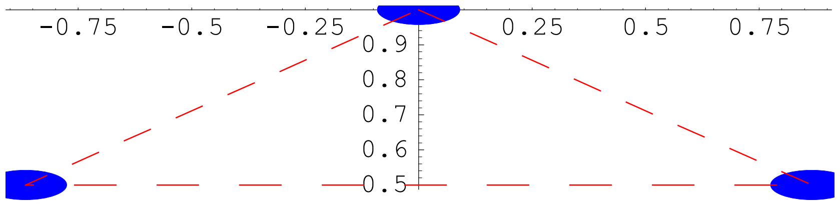
```
In[136]:= ListPlot[XYdata, PlotStyle → {RGBColor[1, 0, 0], Dashing[{0.05, 0.05}]},
           PlotJoined → True, Axes → False];
          (*  Add color, dashed lines, and remove axes  *)
```



```
In[137]:= ListPlot[XYdata, PlotStyle → {RGBColor[0, 0, 1], PointSize[0.1]}];
          (*  Do not connect points  *)
```

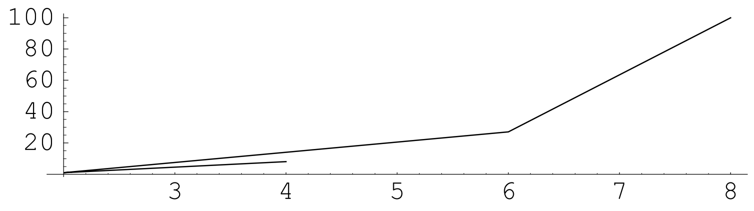


```
In[138]:= Show[%, %%]; (*  Put two previous plots (% and %%) together  *)
```



Note, to make the geometric figure without internal lines, the order of points matters:

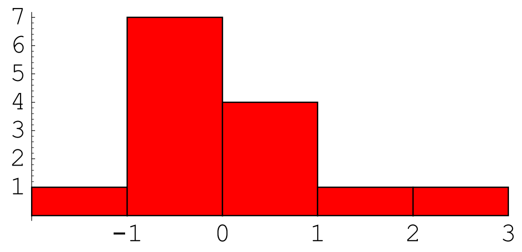
```
In[139]:= plotdata = {{4, 8}, {2, 1}, {6, 27}, {8, 100}} (*  Out of order points  *)

Out[139]= {{4, 8}, {2, 1}, {6, 27}, {8, 100}}

In[140]:= ListPlot[plotdata, PlotJoined → True] ; (*  Out of order plot  *)
```



■ 4.10.4 Numerical Plots: Histograms

```
In[141]:= << Statistics`DescriptiveStatistics`

In[142]:= data = {-2., -0.8, 2., .0, -.5, -.5, 1.6, .8, .5, -.5, -.2, -.2, .2, -.1}

          General::spell1 : Possible spelling error: new
             symbol name "data" is similar to existing symbol "Ydata".

Out[142]= {-2., -0.8, 2., 0., -0.5, -0.5, 1.6, 0.8, 0.5, -0.5, -0.2, -0.2, 0.2, -0.1}
```

```
In[143]:= Histogram[data]
```



```
Out[143]= - Graphics -
```

■ 4.10.5 Surface Plots of Data: `ListPlot3D`

In Section 4.5 we described how to make 3-D or surface plots of analytic functions of two variables, *s = f(x, y)*. Here we describe how to make the same kind of plot when there is no analytic function *f(x, y)*, but just a table of numbers. In Chapter 13 we describe how to use the free plotting program *gnuplot* to make surface plots of numerical data. (Given a choice, we would recommend *gnuplot*.)

Most often, realistic calculations produce numerical data rather than analytic functions as their output. This is a consequence of the real world being less simple than the assumptions made in those models which lead to purely analytic answers. In realistic cases, the resulting equations can still be solved, only they must be solved numerically. Trying to understand if there is meaning in long lists of numbers obtained as output can be quite the challenge, yet this is exactly where the visualization tools are most valuable.

As was the case with analytic functions, a table or list of numbers may represent a function of one variable, a function of two variables, or even more. While this may sound absolutely dreadful to unravel, the use of matrices and arrays makes the bookkeeping rather straightforward. While we have not yet discussed how *Mathematica* goes about handling matrices (we do that in Chapter 7), we will use some of those concepts here. Accordingly, you may want to read about matrices and then return to this subsection.

As with all 3-D visualizations, we want to create a surface in a three dimensional space that represents our data. That's what we did when we plotted the temperature *T[x,y]* as a height, with its functional dependences on *x* and *y* displayed by moving to different regions in the *x y* plane. To be more specific, we can imagine the surface that we are creating is described by a height *z[x,y]* that varies for different positions *x* and *y* in a plane. While it is conventional to draw our 3-D plots so that *z* corresponds to the vertical direction above the *x y* plane, once you have your plot, you can rotate it around and, like a pilot flying upside-down too long, you may forget which end is up.

Let us imagine now that we are dealing with data describing the temperature *T(x)* as a function of distance *x* along a one dimensional bar. However, the bar is cooling as a function of time and so there are a number of these temperature distributions, each for a different time. It is neat and convenient to place all the data together in a function of both time and position, *T(t, x)*. Note that the bar has only one spatial dimension, the distance *x*, yet time *t* is also a variable so we must visualize a function of two variables. We make a graph of *T(t,x)* with *T* as a vertical distance, with time *t* as a horizontal coordinate, and with position *x* as a different horizontal coordinate. The fact that a 3-D surface of this nature does not truly exist in nature is one reason this approach is called ``visualization".

The plotting of numerical data is done in two steps:

1) read the data into a matrix with one index for position and one for time

2) have *Mathematica* plot the matrix

We have placed the data we want to visualize in the file `EqHeat_z.dat` on the CD. The file should be what is called ``plain, flat, text or ASCII", that is, one without control characters. Now make sure that file has been moved to the current working directory for *Mathematica*, or to a place where it can be easily found.

Why Just *z* Values for Plot?

It may appear obvious that if we want to plot the function *z(x,y)*, then the input data file should contain *x* and *y* values and the associated *z* value. While this might be the most direct approach to describing a function, keeping track of all those numbers can get rather complicated, as well as requiring you to deal with lots of numbers. (However, this is fine for analytic functions, examples of which are found in the section on `ContourPlot3D`.) For this reason, *numerical 3-D plots are usually created using just a set of z values as input*, with the assumption that these *z* values correspond to a rectangular array of uniformly spaced *x* and *y* values. This is, in fact, the reason we have placed the subscript *_z* in the file name `EqHeat_z.dat`, it is a reminder that the file contains just *z* values.

Go take a look at the file `EqHeat_z.dat` with a text editor (not *Mathematica*). You should see that it contains lines of numbers (11 numbers on each line) separated by spaces. The first three lines should look like this

```
0.0     100.0  100.0  100.0  100.0  100.0  100.0  100.0  100.0  100.0  0.0
```

```
0.0    32.5   59.8   78.9   89.5   92.8   89.5   78.9   59.8   32.5   0.0

0.0    22.7   43.0   59.0   69.1   72.5   69.1   59.0   43.0   22.7   0.0
```

Think of each line of data as a row of a big matrix. Even though the number of digits used for each number may differ, and so the length of each line may differ, each row has the same number of entries or columns. These data correspond to values of the temperature for increasing *x* values along the rod. Although explicit values of the time are not given, subsequent lines (rows of the matrix) contain the temperature distributions for later and later times. In other words, the full matrix is

```
T[[x, t]] = T[[column, row]]
```

where each new row corresponds to the next value of the time. Since no explicit values are input for the time or the position, the plotting program will assume uniform spacing and time steps, and will assign each integer values, *1, 2, 3 ...* for the plot. You can think of these as ``the first, second, *et cetera* times" and the ``first, second, *et cetera*" positions.

Reading in Just *z* Values of Data

The `Import` command is very handy for importing data from files. We view each row of the matrix (on separate lines) as a list :

```
{value_1, value_2, ..., value_numcols}
```

and the collection of rows as a list of lists:

```
{{row 1 data}, {row 2 data},  ... {row numrows data}}
```

In *Mathematica* we do not need to know how many rows or columns there are to build the matrix. The first argument for Import is the path and name of the file you wish to import data from. The second argument allows you to specify the format you wish the data to be stored as. If you have a file in which each line consists of a single number, then you can use `Import["file","List"]` to import the contents of the file as a list of numbers. We, however, have a file where each line consists of a list of numbers separated by spaces, so we want to use `Import["file", "Table"]` command. This will yield the list of lists of numbers that we are looking for. You will learn in Chapter 7 that a list of lists is a matrix in *Mathematica*, also known as a `Table`. If your file has a ".dat" extension you do not have to specify the format in the second argument, the data will automatically be read using `Table` format with the single argument being the file name and extension. Other import formats are available for text, words, graphics, sound, etc. For more information, look up help on the `Import` command.

The data (values of *z*) are read into the variable (object or abstract data type) that we name *data* with *Mathematica's* `Import` command. Since files are stored on your individual computer using its own file system, just how you specify the file name depends somewhat on the computer system you are on, and where the file is on that system. For all but the simple Unix version, we have placed a (* ... *) around the command so that *Mathematica* will treat the command as a comment. Note the use of quotes in the command around the file name as well as around the type of data the file is to be read in as. To see what works for you, try deleting the comment symbols around the command and seeing if the command completes with no error message:

```
In[144]:= (* data =
            Import["/home/robynw/Book/Summer03/Mathematica/EqHeat_z.dat", "Table"] *)
          (*  Unix system, file in specified directory  *)

In[145]:= (*   data = Import["EqHeat_z.dat", "Table"]  *)
          (*  Unix system, file in same directory where Mathematica started  *)

In[146]:= (*   data = Import["Mac G3 HD:DMc:EqHeat_z.dat", "Table"]  *)
          (*  Apple MacIntosh  *)
```

```
In[147]:= (*   data =
             Import["C:\\My Documents\\Rubin\\Books\EqHeat_z.dat", "Table"]  *)
           (*  Windows, needs extra \   *)

In[148]:= (*   data =
             Import["C:/My Documents/Rubin/Books/Intro/Mathematica/EqHeat_z.dat",
              "Table"]  *) (*  Works also on Windows  *)
```

As you can see in the output cell, the variable *data* is a list {...} containing other lists {{...}, {...}, {...}}. Each sublist is the temperature all along the bar at a different time. In case you have some trouble with reading these files, you may also want to try inputting the data by hand with the list of lists format, *data = {{...}, {...}, {...},....}*.

We can look at any individual part of this list that we want. For example, here is the second row of *data* (the second list):

```
In[149]:= data[[2]]

Out[149]= -0.8
```

As a check, let us look at some individual elements. For example, here's the 5th element in the 2nd row:

```
In[150]:= data[[2, 5]]

          Part::partd : Part specification ≪1≫ is longer than depth of object.

Out[150]= {-2., -0.8, 2., 0., -0.5, -0.5, 1.6, 0.8, 0.5, -0.5, -0.2, -0.2, 0.2, -0.1}⟦2, 5⟧

In[151]:= (*  Determine the element in row 5, column 2  *)
```

4.10.6 ListPlot3D

The `ListPlot3D` command creates a 3-D plot of a surface representing a list of lists of numeric values. That is, a visualization of a 2-D matrix. The options are much the same as those for `Plot3D`. We give an example in which a surface is drawn from our imported data. Rotating the surface is lovely. Try it with the `3D ViewPoint Selector` menu, pasting the option into the command:

```
In[152]:= ListPlot3D[data, ColorFunctionScaling → True,
            ColorFunction → Hue , ViewPoint -> {-2.343, 2.209, 1.040}];

          SurfaceGraphics::gmat :
           {-2., -0.8, 2., 0., -0.5, -0.5, 1.6, 0.8, 0.5, -0.5, -0.2, -0.2, 0.2, -0.1}
             is not a rectangular array larger than 2 x 2.
```

4.11 Plotting a Matrix: `ListPlot3D`

As we have just shown above, the way to make a 3-D plot of a matrix is with the `ListPlot3D` command in *Mathematica*. Unlike Maple, *Mathematica* does not have a *matrixplot* command to do it all in one step. When we read in the data from the file, it was stored as a `Table` format, which is a matrix in *Mathematica,* therefore no conversion from a list to a matrix is necessary. We verified it was a matrix by accessing the individual rows and elements in the usual way. In Chapter 7 we discuss using matrices in *Mathematica*, and give examples of plotting matrices and vectors as well. If you have trouble following the procedure below, you may want to learn some more about matrices in Chapter 7.

```
In[153]:= slant = {{1, 5, 9}, {2, 6, 10}, {3, 7, 11}, {4, 8, 12}};
          slant // MatrixForm
          (*  Set up matrix with three columns  *)

  Out[154]//MatrixForm=
```

$$\begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{pmatrix}$$

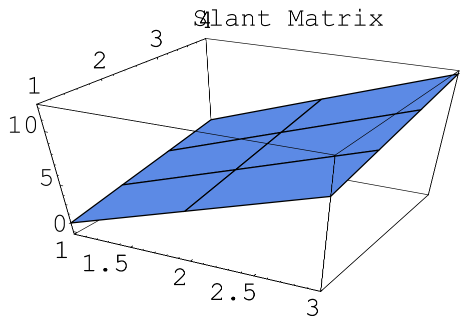
```
In[155]:= ID = IdentityMatrix[6]; (*  The identity matrix of specified dimension  *)
          ID // MatrixForm

  Out[155]//MatrixForm=
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```
In[156]:= ID2 = IdentityMatrix[3];
          ID2 // MatrixForm

  Out[157]//MatrixForm=
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
In[158]:= ListPlot3D[slant, BoxRatios → {1, 1, 1}, PlotLabel → "      Slant Matrix"];
          (*  A new slant  *)
```



For our data above, different rows correspond to different values for the time *t*, and different columns correspond to different positions *x* along the bar. These are plotted along the *x* and *y* axis in the plot, that is along the base of the solid figure. The temperature is plotted as the height of the surface above the base. To help with the visualization, we will also use the `Lighting->False` option to color the surface with gray scales determined by height, white being hot and black being cold, to convey the temperature:

```
In[159]:= ListPlot3D[data, Lighting → False, ViewPoint -> {-2.343, 2.209, 1.040}];

          SurfaceGraphics::gmat :
           {-2., -0.8, 2., 0., -0.5, -0.5, 1.6, 0.8, 0.5, -0.5, -0.2, -0.2, 0.2, -0.1}
             is not a rectangular array larger than 2 x 2.
```

We can also color the surface with the `ColorFunction->Hue` option to yield a range of colors. However, this doesn't help up visualize the hot and cold temperature heights as well:

```
In[160]:= ListPlot3D[data, ColorFunctionScaling → True,
            ColorFunction → Hue , ViewPoint -> {-2.343, 2.209, 1.040}];

          SurfaceGraphics::gmat :
           {-2., -0.8, 2., 0., -0.5, -0.5, 1.6, 0.8, 0.5, -0.5, -0.2, -0.2, 0.2, -0.1}
             is not a rectangular array larger than 2 x 2.
```

ListPlot3D is similar to other 3-D surface plots. You already have some experience with Plot3D and so you should know how to label the axes and plot, as well as rotate the surface.

- Label the axes indicating *row* and *column*. Recall that increasing *row* numbers correspond to increasing time, while the increasing column numbers correspond to increasing position along the bar.
- Rotate the plot using the 3DViewPointSelector under the Input menu to paste the option in the command.
- Add a title to the plot and remove the grid mesh lines with the Mesh option. Do you think the mesh grid helps give the 3-D impression?

You have the figure above, which is rather complete, but not necessarily an effective visualization. You are probably the best judge of effectiveness, and to do that you need to try out the options and see what works.

Reading in (*x, y, z*) Data Sets

In a more realistic case our data might be the results of a numerical simulation or a measurement and would reside in an external file. We read in these data as we did before with the Import command, but now with 3 columns for the values of (*x*, *y*, *z*), still importing the data into a Table structure. For this purpose we have supplied the file *EqHeat_xyz.dat*:

```
In[161]:= datalistex =
           Import["/home/robynw/Book/Summer03/Mathematica/EqHeat_xyz.dat", "Table"]
          (*  Unix read, complete specification  *)

In[162]:= (*  datalistex =
            Import["C:\\My Documents\\Rubin\\Books\\Intro\\Maple\\EqHeat_xyz.dat",
             "Table"]  *)
          (*  Windows read, complete specification  *)

In[163]:= ListPlot3D[datalistex];
```

4.12 Animations of Data*

In the corresponding section in the text is an animated *Mathematica* plot of the wave motion resulting from plucking a string that is hanging under its own weight, and is affected by friction. This comes from a numerical simulation [CP] that outputs its results to a file in the *gnuplot* format used for surface [$z(x, y)$] plots. (As described in Sec. 4.9.5, the data are in the form of a matrix of z values, with the rows of the matrix separated by blank lines. The first row corresponds to time 1, with the place in the row corresponding to different x values. Row two contains all the z values for time 2, and so forth.) As we see in Fig. Waves3D.eps, the surface plot shows many ripples corresponding to oscillations of the string, but is not nearly as effective a visualization as playing the animation below.

In this section we indicate the steps needed to input numerical data in surface plot format, and convert it into an animation. We start by forming a very long list of data named ``data" from the file *function.dat* (Unix reads from present directory, Windows requires full path name as give by Explorer):

```
In[164]:= data =
            Import["/home/robynw/Book/Summer04/Mathematica/function.dat", "List"] ;
           (*  Unix system, file in specified directory, .dat  uses "Table" format  *)

           data[[1]] (*  Individual elements in list  *)
           data[[4]]
```

We next break up the array *data* into a list of sublists. The first sublist *func[1]* corresponds to row 1 of the original matrix, the second sublist, *func[2]* corresponds to row 2 of original matrix, and so forth:

```
fdata[x_] := data[[x]]; (*  Write array as a function  *)
? fdata

func = {Array[fdata, 101, 1]} (*  Extract first row into func  *)
```

We now use the `For` looping structure, which we talk about extensively in Chapter 8, to append additional rows as lists to func to create a list of lists:

```
For[t = 1, t ≤ 199, t = t + 1, {AppendTo[func, Array[fdata, 101, 101 * t + 1]]}];
```

As a check, we print out the sublist *func[1]*. It is the first row of input and will be the first frame of the movie:

```
func[[1]] (*  Print out individual row  *)
```

Now that we know that the data looks good, we plot several of the frames that we will put together to form the movie:

```
ListPlot[func[[1]], PlotJoined → True];
ListPlot[func[[10]], PlotJoined → True];
ListPlot[func[[20]], PlotJoined → True]; (*  Plot individual frames    *)
```

To create the movie, we would like to make a *sequence* (ordered list) of plots and then animate them. We can make an array of all the plots in order, however we have not found a way to display them all on one plot, (the `Show` command would be the obvious way to do it, but doesn't seem to work with plots as objects). Instead, *Mathematica* plots all 200 plots. To animate them you need to select the cell containing them all and choose Cell => Animate Selected Graphics. That's why here we plotted several of the rows.

```
ListPlot[func[[1]], PlotJoined → True];
ListPlot[func[[10]], PlotJoined → True];
...
ListPlot[func[[190]], PlotJoined → True];
```

Animate them as explained above. Now that is a movie! Control the speed and direction of the animation with the mouse buttons in the bottom left of the window.

4.13 Key Words and Concepts

abstract data type abscissa animation contour plots dependent variable implicit plot independent variable list matrix nonlinear functions ordinate set sequence surface plot 2D plot parametric plot polar plot potential energy

1. Does potential energy occur in nature?
2. Is there a reason for electric charge to occur always in integer values?
3. How does an abstract data type differ from an algebraic symbol?
4. How do you decide which is an *independent* and which is a *dependent* variable?
5. When might it be a bad idea to use color in your plotting?
6. List three ways in which you may change the apparent meaning of data by changing the way in which it is represented.
7. What might be a *dishonest* way of presenting your data?
8. Give examples of the type of data that may be appropriate for 1D, 2D, 3D, and 4D visualizations.
9. When are animations a useful way to display data?
10. How, in a mathematical sense, does a phase-space (parametric) plot differ from an ordinary 2D plot?
11. How is a table of numerical data similar to, and different from, a mathematical function?

4.14 Further Exercises

1. On a single graph, display the function x^3 Sin(x), x^3 Cos(x), and x Log*(x)*, each in a different color. Use an equal negative and positive x range, and pick that range to obtain the most interesting comparison of the three functions.

2. A graphical approach to solving equations plots the right and left hand sides of an equation as two separate functions, and then shows the solution to the equation as the value of the abscissa at which the two functions intersect. In other words, a solution of *f(x)=g(x)* occurs when the graphs of *f(x) versus x* intercepts the graph of *g(x) versus x*. Use *Mathematica's* ability to plot several functions in the same graph to determine the approximate solutions of the following equations

(a) $\text{Sin}[x] = x^2$
(b) $x^2 + 6x + 1 = 0$
(c) $H^3 - 9H^2 + 4 = 0$

3. Do a graphical experiment in which you prove to yourself these very useful mathematical facts:

a) the exponent e^x grows faster than any power x^n

b) the logarithm ln(x) grows slower than any power x^n

Hint: To avoid overflow problems with very large *x* values, you may want to make semilog plots.

4. Do a graphical experiment to find the value of *n* for which these equations are true:

a) $n\ \mathrm{Sin}(2x) = \mathrm{Sin}(x)\ \mathrm{Cos}(x)$

b) $n\ (\mathrm{Cos}(x))^2 = 1 + \mathrm{Cos}(2x)$

5. If two tones very close in frequency are played together, your ear hears them as a single tone with oscillating amplitude. Make plots as a function of time of the results of adding the two sine functions

$$\mathrm{Sin}(100\,t) + \mathrm{Sin}(b\,t)$$

Make a series of plots for *b* in the range

$$90 < b < 100$$

Make sure to plot for a long enough range of *t* values to see at least three cycles of any periodic behavior.

6. Here are nine measurements given in the form (*x, y*):

(0,10.6), (25, 16.0), (5, 45.0), (75, 83.5), (100, 52.8), (125, 19.9), (150, 10.8), (175, 8.25), (200, 4.7)

Make a plot of these data points.

7. The orbits of planets and comets are known to be conic sections. Conic sections are the 2-D curves formed when a cone is cut (sectioned) by a plane, and are given by the parametric equations in which *s* is the parameter:

a) hyperbola: $(x(s), y(s)) = (4\ \mathrm{Cosh}(s), 1.4\ \mathrm{Sinh}(s))$

b) ellipse: $(x(s), y(s)) = (4\ \mathrm{Cos}(s), 1.4\ \mathrm{Sin}(s))$

c) parabola: $(x(s), y(s)) = (s\ \mathrm{Cos}(\theta) - s^2\ \mathrm{Sin}(\theta), s^2\ \mathrm{Cos}(\theta) + s\ \mathrm{Sin}(\theta))$, θ = arbitrary parameter.

Make parametric plots of these conic sections. Cover as much range as is needed for the parameter *s* in order to obtain the familiar shapes.

8. In polar coordinates, the conic section is described by the equation

$$\frac{\alpha}{r} = 1 + \epsilon\,\mathrm{Cos}[\theta]$$

where ϵ is the eccentricity and 2α is the latus rectum of the orbit. An ellipse occurs when $0 < \epsilon < 1$, a hyperbola for $\epsilon > 1$, and a parabola for $\epsilon = 1$. Make polar plots of these three kinds of orbits based on the polar equation of the conic section. Try various values for the parameter α (it is inversely proportional to the energy of the planet).

9. Make a surface plot of the Yukawa potential:

$$V(x, y) = \frac{e^{(-r)}}{r}\,\mathrm{Cos}\left[\frac{x}{r}\right],\ r = \sqrt{x^2 + y^2}$$

10. Make a contour plot of this same Yukawa potential.

11. A standing wave is described by the equation

$$y(x, t) = \mathrm{Sin}[10\,x]\,\mathrm{Cos}[12\,t]$$

where *x* is the distance along the string, *y* is the height of the disturbance, and *t* is the time.

a) Create an animation of this function.

b) Create a surface plot of this function, and see if you agree with us that it is not as revealing as the animation.

12. A traveling wave is described by the equation

$$y(x, t) = \mathrm{Sin}[10\,x - 12\,t]$$

where x is the distance along the string, y is the height of the disturbance, and t is time.

a) Create an animation of this function.

b) Create a surface plot of this function, and see if you agree with us that it is not as revealing as the animation.

13. Give an example or two of the type of function(s) that would be visualized best with each of the following plots:

a) 2-D plot

b) 3-D plot

c) multifunction plot

d) parametric plot

e) animation

f) 3-d animation

14. Explain in just a few words what is meant by

a) an abstract data type

b) a parametric or phase space plot

c) a function of three variables

d) a list of three variables

15. Plot the function $f(x) = Sec(x) + 4$ over the interval $[0,4\,\pi]$. Since *Mathematica's* automatic scaling does not work well here, you will need to specify a range for the ordinates to obtain a very useful visualization.

16. Given the points (1, 0.53), (1.5, 0.65), (2, 0.91), (2.5, 0.95), and (3, 1.10). Create a plot containing these points as well as the functions

$$a(x) = \mathrm{Sin}\left[\frac{x}{2}\right], \quad b(x) = \frac{x^2}{5},$$

and thereby determine which function fits the data better?