Maple for Math Majors

Roger Kraft
Department of Mathematics, Computer Science, and Statistics
Purdue University Calumet
roger@purduecal.edu

5. Graphs of functions and equations

5.1. Introduction

We define many kinds of graphs in mathematics and Maple has a lot of commands for drawing different kinds of graphs. In the next section we describe nine kinds of graphs commonly used in calculus courses and seven Maple commands that are used to draw these graphs. The goal of this worksheet is to help you to understand all of these different kinds of graphs and to be able to work with them in Maple.

In the following section we review exactly what we mean by the graph of a function and the graph of an equation. By looking carefully at these definitions of a graph, we see that graphs and functions can be classified in certain ways. Maple uses these classifications to organize its graphing commands, so understanding Maple's graphing commands boils down to understanding how we can classify graphs and functions. The classification of functions and graphs can also help you to understand other ideas in mathematics, for example, all the different kinds of derivatives that you learned about in calculus (e.g., ordinary, partial, implicit, tangent vector, gradient vector, Jacobian).

[>

5.2. A review of graphs

Maple has a lot of graphing commands. This is because Maple can draw a lot of different kinds of graphs. To understand Maple's graphing abilities it helps to have a way of classifying graphs so that we can organize Maple's graphing commands according to what kind of graph they draw. There are several ways of classifying graphs. For example, there are two dimensional graphs vs. three dimensional graphs, or graphs of equations vs. graphs of functions. In addition, graphs of functions can also be classified by how we draw the graph. For example we can graph inputs vs. outputs, or we can graph just the outputs, or we can graph a vector field. We will explain and use all of these classifications to help us understand graphs in general.

[>

From Maple's point of view, the main way of classifying graphs is either as two dimensional or three dimensional. Here is a list of the principle kinds of two and three dimensional graphs that Maple can draw (this is not a complete list but it is enough to get us started).

[>

In two dimensions, Maple can draw the following kinds of graphs:

1) The graph of a real valued function of one real variable.
2) The graph of a parametric curve in the plane
   (that is, the graph of a 2-dimensional vector valued function of one real variable).
3) The graph of a vector field in the plane
   (that is, the graph of a 2-dimensional vector valued function of two real variables).
4) The graph of an equation in two real variables.

In three dimensions, Maple can draw the following kinds of graphs:

5) The graph of a real valued function of two real variables.
6) The graph of a parametric curve in 3-dimensional space
   (that is, the graph of a 3-dimensional vector valued function of one real variable).
7) The graph of a parametric surface in 3-dimensional space
   (that is, the graph of a 3-dimensional vector valued function of two real variables).
8) The graph of a vector field in 3-dimensional space
   (that is, the graph of a 3-dimensional vector valued function of three real variables).
9) The graph of an equation in three real variables.

Notice that the list includes two kinds of equations and seven kinds of functions. Before going into the details of drawing these kinds of graphs with Maple, we should look at an example of each one of them. But before doing even that, let us review the idea of the graph of a function and the graph of an equation.

[**>**

Recall that an equation is made up of an equal sign with an expression on either side of it. Equations ask a question. Does the left hand side have the same value as the right hand side? If the expressions have unassigned variables in them, then we can ask which values of the unassigned variables make the equation true (that is, what values of the variables solve the equation). The **graph of an equation** is a plot of those values for the unassigned variables that make the equation true. If the equation is an equation in one variable, then its graph is made up of points in the real line (which is not visually interesting, and Maple does not even have a command to "draw" such a graph). If the equation is an equation in two variables, then its graph is made up of points in the plane and the graph is usually, but not always, a curve in the plane. If the equation is an equation in three variables, then its graph is made up of points in 3-dimensional space and the graph is usually, but not always, a two dimensional surface. If the equation has four (or more) variables, then its graph is made up of points in four (or more) dimensional "space". This is something that we cannot easily visualize and Maple cannot draw (at least not directly), but it is important to realize that graphs of equations do in fact exist even in these higher dimensional cases.

[**>**

Recall that a function is made of three things, a set of inputs (the domain), a set of outputs (the codomain), and a rule for associating one output to each of the inputs. The **graph of a function** is a visualization of the relationship between the inputs and outputs of the function. Since a function is

made of three things, so is its graph. The graph has a visualization of the input set, a visualization of the output set, and a visualization of the rule. If a function has an *m*-dimensional input and an *n*-dimensional output, then the graph of the function is a plot in $(m + n)$-dimensional space with the *m*-dimensional input axes perpendicular to the *n*-dimensional output axes and a point plotted for every combination of an input with its output. This is a bit cumbersome to describe in words. The most important thing to remember is that the graph of a function is drawn in a space whose dimension is the sum of the dimensions of the inputs and the outputs. We sometimes refer to this as an **input vs. output** kind of graph.

For the purpose of drawing graphs with Maple, we will consider three kinds of sets of inputs and outputs. An input or output set will be either the 1-dimensional real line, the 2-dimensional plane, or 3-dimensional space. That gives us nine kinds of functions to worry about, i.e., all possible combinations of the three kinds of input sets with the three kinds of output sets. But for some of these kinds of functions, trying to draw its graph creates a problem. If *m* and *n* are the dimensions of the inputs and the outputs and $m + n$ is more than three, then how do we draw the graph? This problem comes up for six of the nine possible kinds of functions we have to worry about (which ones are they?). We solve this problem by considering two other kinds of graphs for a function besides the graph which plots inputs vs. outputs.

[**>**

One of these other kinds of graphs, a **parametric graph**, visualizes a function by plotting only the outputs of the function. This type of graph will be used for three kinds of functions, functions from the line to the plane, functions from the line to space, and functions from the plane to space. Notice that for two of these kinds of functions, an input vs. output kind of graph is not possible (why?). But for functions from the line to the plane, while an input vs. output graph is possible, it turns out that a parametric graph has traditionally been more useful.

The third kind of graph that we will consider is a vector field. A **vector field** is kind of an input vs. output graph, but the input axes are not shown perpendicular to the output axes. Instead the inputs and the outputs are draw on the same set of axes. This only works when the function has the same number of inputs as outputs. So we draw vector fields for functions from the plane to the plane (2-dimensional vector fields) and for functions from space to space (3-dimensional vector fields). (It is possible, and useful, to draw a 1-dimensional vector field, but Maple does not have a command for this.)

If you have been following along very carefully, you know that there are still two kinds of functions that we have not yet considered. Functions from space to the line (real valued functions of three real variables) and from space to the plane (2-dimensional vector valued functions of three real variables) do not have any practical way of being visualized and Maple does not have any commands for visualizing them, so we will not consider these two cases any further. (This does not imply that these kinds of functions are not useful or important, just that Maple does not have a way of visualizing them.)

[

[>

So we end up with seven kinds of functions to graph. Two of these kinds of functions will have input vs. output graphs and will require two different Maple commands, **plot** and **plot3d**. Three kinds of functions will have parametric graphs and will require three different Maple commands, **plot**, **spacecurve**, and **plot3d**. And two kinds of functions will be graphed as vector fields which will require two Maple commands, **fieldplot** and **fieldplot3d**. If we add in the two kinds of graphs of equations, equations in two variables and equations in three variables, which require two more Maple commands, **implicitplot** and **implicitplot3d**, then we have a total of nine kinds of graphs and Maple has seven different commands for drawing these nine kinds of graphs!

Now that we have briefly reviewed the idea of graphing functions and equations, let us look at an example of each of the nine kinds of graphs that we have mentioned. Each kind of graph will turn out to have its own Maple syntax. Without the classification we made above, all these different commands and syntaxes can become pretty confusing. Hopefully, understanding the kind of function we are given and the kind of graph that we want to draw will help us to remember the proper command to use.
[>

First the four kinds of two dimensional graphs. For each of these graphs, be sure to click on the graph and then play with some of the buttons from the graphics context bar, examine the menu items from the graphics menu bar. and also try right clicking on the graph and experimenting with the graphics context menus.

1) The graph of a real valued function of one real variable. These are the functions that you are most familiar with from the first two semesters of calculus. These are also the most common examples of input vs output graphs. The following command graphs the function $x \to x + 2\sin(x)$.

```
[ > plot( x+2*sin(x), x=-10..10 );
[ >
```

2) The graph of a parametric curve in the plane (that is, the graph of a 2-dimensional vector valued function of one real variable). These come up in second semester calculus and also in physics. They represent the path of motion of something moving in the plane. This is an example of where a function's parametric graph (output only graph) is more useful than its input-output graph (which could be drawn; why?). The following command graphs the function $t \to (t\sin(t), t\cos(t))$.

```
[ > plot( [t*sin(t), t*cos(t), t=0..3*Pi] );
```

Notice that the two formulas (why are there two?) are put in a list along with their range. It is the fact that the range is inside the list that makes this a parametric graph of a single (vector valued) function and not a graph of two (real valued) functions. Try moving the range outside of the list and see what happens (do not forget to put a comma between the list and the range).

[

```
> 
```

3) The graph of a vector field in the plane (that is, the graph of a 2-dimensional vector valued function of two real variables). These functions come up in differential equations courses and also in vector calculus. In those settings a 2-dimensional vector valued function of two variables is visualized as a vector field. Maple has a special command for graphing a vector field. Notice that a vector field is neither an input-output nor a parametric type of graph. The following command graphs the function $(x, y) \to \left(\frac{y}{\sqrt{x^2+y^2}}, \frac{x}{\sqrt{x^2+y^2}} \right)$.

```
> plots[fieldplot]( [y/sqrt(x^2+y^2), x/sqrt(x^2+y^2)], x=-5..5,
  y=-5..5 );
```

```
> 
```

4) The graph of an equation in two real variables. The simplest examples of these are the equations for the conic sections (i.e., hyperbola, parabola, ellipse). Here is a more complicated example of an equation and it graph. The following command graphs the equation $8\,x^3 + 8\,y^3 = 1 + 40\,x\,y$.

```
> plots[implicitplot]( 8*x^3+8*y^3=1+40*x*y, x=-3..3, y=-3..3 );
```

```
> 
```

In three dimensions, Maple can draw the following five kinds of graphs: For each of these graphs, be sure to click on the graph and try rotating it. Also, play with all the buttons in the 3D graphics context bar and examine the menu items in the 3D graphics menu bar.

5) The graph of a real valued function of two real variables. These are the functions that are first studied in third semester calculus. These are input vs. output graphs. The following command graphs the function $(x, y) \to x^2 + y^2$.

```
> plot3d( x^2+y^2, x=-2..2, y=-2..2 );
```

```
> 
```

6) The graph of a parametric curve in 3-dimensional space (that is, the graph of a 3-dimensional vector valued function of one real variable). These come up in third semester calculus and also in physics. They represent the path of motion of something moving in space. This is another example of a kind of function where a parametric graph is more useful than an input vs. output graph (the input-output graph would need four dimensions anyway). The following command graphs the function $t \to (t\sin(t), t\cos(t), t)$.

```
> plots[spacecurve]( [t*sin(t), t*cos(t), t], t=0..9*Pi );
```

```
> 
```

7) The graph of a parametric surface in 3-dimensional space (that is, the graph of a 3-dimensional vector valued function of two real variables). These come up at the end of third semester calculus and in vector calculus courses. (How many dimensions would the input-output graph of one of these functions need?) The following command graphs the function
$(u, v) \to ((2 + \sin(v)) \cos(u), (2 + \sin(v)) \sin(u), u + \cos(v))$.

```
> plot3d( [(2+sin(v))*cos(u), (2+sin(v))*sin(u), u+cos(v)],
>          u=-2*Pi..2*Pi, v=-2*Pi..2*Pi );
```

Try changing the brackets (**[** and **]**) to braces (**{** and **}**). Can you explain the resulting graph?

```
>
```

8) The graph of a vector field in 3-dimensional space (that is, the graph of a 3-dimensional vector valued function of three real variables). These functions come up in differential equations and vector calculus courses. In those settings a 3-dimensional vector valued function of three variables is visualized as a vector field. Maple has a special command for graphing vector fields in 3-dimensional space. Notice again that a vector field is neither an input-output nor a parametric type of graph. The following command graphs the function $(x, y, z) \to (2x, 2y, 1)$.

```
> plots[fieldplot3d]( [2*x,2*y,1], x=-1..1, y=-1..1, z=-1..1,
>                      grid=[5,5,5] );
```

```
>
```

9) The graph of an equation in three real variables. The simplest examples of these are the quadric surfaces that are studied in second or third semester calculus. But here is a far more complicated example. The following command graphs the equation $x^3 + y^3 + z^3 + 1 = (x + y + z + 1)^3$.

```
> plots[implicitplot3d]( x^3+y^3+z^3+1 = (x+y+z+1)^3,
>                        x=-2..2, y=-2..2, z=-2..2 );
```

```
>
```

Finally, notice two things. First, the **plot** and **plot3d** command each handle two kinds of graphs (which are they?). That is why there are seven Maple commands for drawing nine kinds of graphs. Second, the other five of these seven Maple commands are contained in the **plots** package. We could load the **plots** package at this time and save ourselves some typing later in this worksheet, but we will not do that. Instead, we will always use the long name for these commands. This has two advantages. First, it helps to remind us that these commands are from the **plots** package and second, all the commands in this worksheet will always work, regardless of where in the worksheet you might jump in and start working from.

```
>
```

Exercise: In all of the examples above, the mathematical functions were represented by Maple expressions. Try modifying each example to use a Maple function.

[>

Exercise: Part (a) For each of the seven kinds of functions described above, describe what kind of derivative is appropriate for that type of function. Give your answers in terms of ideas from calculus, not in terms of Maple.

Part (b) For each of the seven kinds of functions described above, explain how you might integrate a function of that type. Again, give your answers in terms of ideas from calculus, not in terms of Maple.

Part (c) What about the two kinds of equations? Can you differentiate an equation? Can you integrate an equation?

[>

[>

5.3. Graphs of real valued functions of one variable

Maple's most basic graphing command is **plot**. This command draws graphs of real valued functions of one real variable. We graph such functions by giving **plot** the function and a range for the independent variable. Here is an example.

```
> plot( sin(x)/x, x=-5*Pi..5*Pi );
```

When **plot** draws the graph, it gives the horizontal axis the range we specified in the range for the independent variable. For the vertical axis, **plot** decides for itself what is an appropriate range. Usually the vertical range is from the minimum to the maximum of the function (as in the above graph). This can sometimes cause a graph to be a bit strange. For example, the next graph seems to be showing a function that has the horizontal axis as a horizontal asymptote. But look very carefully at the vertical axis. The vertical range is from 2 to 3. The piece of the vertical axis from 0 to 2 is missing!

```
> plot( 2+exp(-x^2), x=-10..10 );
```

If we want to, we can give **plot** a range for the dependent variable instead of letting **plot** determine the vertical range for itself. The next command fixes the ambiguity of the last graph by specifying a vertical range. Notice that we now see very clearly that the horizontal axis is not the horizontal asymptote for this function.

```
> plot( 2+exp(-x^2), x=-10..10, 0..3 );
```

(Notice that the second range did not have a variable associated with it.)

[>

Using a range for the dependent variable is especially useful when graphing a function that has vertical asymptotes. In the next graph, **plot** tries, by default, to accommodate the full range of the dependent variable within the graph. But this range is infinite because of the vertical asymptote at $x = 1$. So the graph is not very useful.

```
> plot( 1/(x-1), x=-1..3 );
```

We can fix the graph, and get a useful picture of the function, by providing a range for the dependent variable.

```
> plot( 1/(x-1), x=-1..3, -20..20 );
```

Notice the vertical red line at $x = 1$. This is *not* Maple's way of drawing the vertical asymptote. This vertical line is an artifact of the way the **plot** command works. We will explain in the next worksheet why this line is there. We can make it go away by using the **plot** option **discont=true**.

```
> plot( 1/(x-1), x=-1..3, -20..20, discont=true, color=red );
```

The option **discont** is an abbreviation for discontinuity.

```
>
```

When using the **plot** command, it is important to make a distinction between the *range* and the *scale* of an axis. The range of an axis is what we have been discussing in the last few examples. It is the part of the number line displayed on the axis. The scale of an axis is how much actual length is assigned to one unit of the axis. Here is an example.

```
> plot( sin(x)/x, x=-3*Pi..3*Pi );
```

In this last graph, the range of the horizontal axis is from -3π to 3π (about –9.425 to 9.425). The range of the vertical axis is from –0.2 to 1. So the range of the horizontal axis is much more than the range of the vertical axis. On the other hand, the scale of the horizontal axis is much less than the scale of the vertical axis. Each unit on the horizontal axis (say from 0 to 1 on the horizontal axis) is much shorter in length than a unit on the vertical axis (say from 0 to 1 on the vertical axis). The **plot** command did not use the same scale on the two axes. In general, once the ranges of the two axes have been determined, the **plot** command will choose scales for each axis so that the final graph is "well proportioned". The scales are usually chosen so that the graph is roughly a square. It is possible to force the **plot** command to use the same scale on both of the axes by using the **scaling=constrained** option to **plot**. Here is the last graph redrawn with the same scale on both axes.

```
> plot( sin(x)/x, x=-3*Pi..3*Pi, scaling=constrained );
```

If you compare the last two graphs, the horizontal scale is about the same in both graphs. It was the vertical scale that was modified in the last graph. Sometimes the **scaling=constrained** option helps the appearance of a graph and often it does not. A quick way to see the difference between constrained and unconstrained scaling is to click on a graph and look at the 2-D graphics context bar at the top of the Maple window. You will see a button in the context bar labeled **1:1**. This button switches the constrained scaling on and off. Try it on the last graph.

```
>
```

When graphing real valued functions of one variable, it is important to realize that for many functions, no one graph can tell us everything we may need to know about the function. So the most important skill to develop when using the **plot** command is modifying the ranges on the horizontal and vertical axes to show different pieces of information about a function. Let us look at some examples and exercises about manipulating the ranges of a graph.

Consider the following rational function and a few of its graphs. Each graph will tell us something different about this function.

```
> f := (1-x^2)/(x-2);
```

The following graph from minus infinity to infinity tells us that the function has a vertical asymptote, two x-intercepts, and that the graph goes to minus infinity as x goes to plus infinity and it goes to plus infinity as x goes to minus infinity.

```
> plot( f, x=-infinity..infinity );
```

Now notice that the next graph tells us a bit more and a bit less. This graph is over a very large domain. It tells us exactly how the graph goes to plus and minus infinity. The graph is skew asymptotic to the line $y = -x$. But this graph obscures the fact that the function has two zeros, and even the vertical asymptote is a bit vague.

```
> plot( f, x=-100..100, -100..100 );
```

The next graph allows us to see where the x-intercepts are and to approximate the local minimum between these intercepts.

```
> plot( f, x=-1.1..1.1 );
```

And the next graph gives us a good approximation of the local maximum to the right of the vertical asymptote.

```
> plot( f, x=2..8, -10..-7 );
```

It would be difficult for any one graph to convey all of the information that the above graphs show. After some trial and error, the following graph shows quite a bit about the function, except for providing good approximations of the x-intercepts and the local maximum and minimum.

```
> plot( f, x=-10..10, -15..15, discont=true, color=red );
```

```
>
```

Exercise: Consider the following function.

$$\frac{x}{x+1} - \cos\left(\frac{40}{x}\right)$$

Part (a) Try to get a sense of what the graph of this function looks like.

```
>
```

Part (b) Use a graph to find the most negative and the most positive x-intercepts for this function. (Hint: The positive answer is quite large.)

```
>
```

Part (c) How many x-intercepts does the graph of this function have between 1/10 and 10?

```
>
```

Part (d) Use a graph to find the most negative local maximum and minimums for this function.

```
>
```

Note: The function in this problem comes up in an interesting "real world" calculus problem. See *Calculus with Maple*, by Frank Hagan and Jack Cohen, and the chapter called "The Asteroid Problem".

```
>
```

Exercise: Find ranges for the `plot` command so that the graph of $\cos(x)$ looks like

a) a horizontal line,
[>
b) a vertical line,
[>
c) a 45 degree line.
[>
Then do the same with the graph of tan(x).
[>

All in all, the **plot** command is not that hard to use. For the most part, one learns to use it by looking at examples. We gave a lot of examples of its use in the first worksheet. In the rest of this section we look at examples of a few more ways of using **plot** that were not shown in the first worksheet.

When we use the **plot** command to graph several functions at the same time, we cannot give each function its own range. For example, in the following graph, suppose we only want to graph the part of the parabola that is contained inside of the semicircle.
[**> plot([x^2, sqrt(1-x^2)], x=-1..1, scaling=constrained);**
To do this, we need to draw graphs for the parabola and semicircle separately, using different ranges in each graph, and then combine the two graphs together using the **display** command from the **plots** package. First let us draw the semicircle.
[**> plot(sqrt(1-x^2), x=-1..1, color=green);**
Now let us give this graph a name for later reference.
[**> graph1 := %;**
Notice how, when we gave the graph a name, Maple returned a large amount of data. This data is Maple's internal description of the graph (it's a PLOT data structure). In another worksheet we will say quite a bit about these PLOT data structures but right now we are not very interested in seeing all of this data. So from now, whenever we want to name a graph, we will use a colon at the end of the assignment command (instead of a semicolon) to suppress the printing of this data. (A quick way to make all of that output go away is to immediately type Ctrl-Z and then skip down to the next command.)

Now let us solve for the two intersection points of the parabola with the semicircle. First the negative intersection point.
[**> a := fsolve(x^2=sqrt(1-x^2), x, -1..0);**
And now solve for the positive intersection point (which, of course, is equal to **-a**).
[**> b := fsolve(x^2=sqrt(1-x^2), x, 0..1);**
Now graph the parabola between these two intersection points.
[**> plot(x^2, x=a..b, color=blue);**
Give this last graph a name (notice the colon at the end of the command).
[**> graph2 := %:**
Now we can combine our two graphs using the **display** command.
[

```
> plots[display]( graph1, graph2, scaling=constrained );
```

There is a way to use a single **plot** command to graph two (or more) functions with each function given a different domain. In the next section we will see how to do this as an application of the parametric form of the **plot** command.

```
>
```

Here is an interesting way to use **plot** together with the **display** command. First, we need to create a special kind of variable, an array.

```
> graphs := array( 1..2, 1..2 );
```

Now let us draw four graphs and use the array variable to name them (recall that we discussed indexed names like these in Worksheet 2).

```
> graphs[1,1] := plot( exp(-x^2)*sin(Pi*x^3), x=-2..2, color=blue
  ):
> graphs[1,2] := plot( exp(-x^2), x=-2..2, color=red ):
> graphs[2,1] := plot( -exp(-x^2), x=-2..2, color=green ):
> graphs[2,2] := plot( [exp(-x^2)*sin(Pi*x^3), exp(-x^2),
  -exp(-x^2)],
>                      x=-2..2, color=[blue,red,green] ):
```

Now combine all four graphs into a two by two array of pictures.

```
> plots[display]( graphs );
```

It is the fact that our four graphs are named by a single array variable that tells **display** to draw them in an array kind of format instead of superimposing all four of them on a single set of axes.

```
>
```

So far we have drawn all of our graphs of real valued functions of one variable with the independent variable running along the horizontal axis and the dependent variable along the vertical axis. Of course, this is a pretty common way to draw graphs but it is not the only way. There are times when it is more convenient to draw a graph the other way, with the independent variable along the vertical axis. For example, we may want to graph the functions $g(y) = y^2 - 1$ or $h(y) = \sin(y)$ with the independent variable along the vertical axis. The following two **implicitplot** commands do this by graphing the *equations* $x = y^2 - 1$ and $x = \sin(y)$.

```
> plots[implicitplot]( x=y^2-1, x=-2..2, y=-2..2 );
> plots[implicitplot]( x=sin(y), x=-1..1, y=-2*Pi..2*Pi );
```

The **plot** command does not make it easy to graph a real valued function of one variable this way. In the next section, and also in the next worksheet, we will see several ways to get the **plot** command to graph a function with the independent variable along the vertical axis. The main point that we want to make here is that the **plot** command gives one of the directions in the Cartesian coordinate system a preferred status. The preferred direction is the horizontal direction and its preferred status is that this is the direction of the axis for the independent variable when graphing a real valued function. We commonly label the horizontal axis using x and the vertical axis using y. Using these labels, the default way for **plot** to graph a function f using Cartesian coordinates is as $y = f(x)$ (and **plot** will not graph $x = f(y)$). We will return to this idea in a later section when we

consider non Cartesian coordinate systems on the plane.

Exercise: Why do the following two **implicitplot** commands draw the exact same graphs as the last two **implicitplot** commands.

```
> plots[implicitplot]( 1=y^2-x, x=-2..2, y=-2..2 );
> plots[implicitplot]( x-sin(y)=0, x=-1..1, y=0..2*Pi );
>
```

Exercise: All of the functions in this section were represented as expressions. Go back through this section and modify the examples to use Maple functions.

```
>
```

```
>
```

5.4. Graphs of parametric curves

Parametric curves are important and usefull. Since there are so many ideas to cover concerning parametric curves, this section is divided into several subsections. The first four subsections go over the basics of drawing parametric curves in the plane and the last two subsections go over the basics of drawing parametric curves in space. Parametric curves in two dimensional space are drawn using a special case of the **plot** command. Parametric curves in three dimensional space have a special command, **spacecurve**, from the **plots** package.

```
>
```

5.4.1. Parametric curves in the plane

A parametric curve in two dimensional space is the output-only graph of a 2-dimensional vector valued function of a real variable. A simple example is a circle, which is the parametric graph of the following function.

$$f(t) = (\cos(t), \sin(t))$$

Notice that this function is defined by two real valued functions of a single variable, the **component functions**. If we think of the function $f(t)$ as describing the position of a particle moving in the plane, then the first component function of $f(t)$ describes the particle's horizontal motion and the second component function describes the particle's vertical motion. In calculus books, this function is often written as a pair of "parametric equations"

$$x(t) = \cos(t) \quad \text{and} \quad y(t) = \sin(t).$$

Here is how we use the **plot** command to draw a circle as a parametric graph. (If the following graph does not look like a circle, click on the graph and then click on the **1:1** button in the graphics context bar.)

```
> plot( [ cos(t), sin(t), t=0..2*Pi ] );
```

Notice that it is the placement of the range for the independent variable inside the brackets that distinguishes this command from the command to draw the graph of two real valued functions. In the next example, we move the range outside the brackets and we get the (input-output) graph of the two real valued functions $\cos(t)$ and $\sin(t)$.

```
> plot( [ cos(t), sin(t) ], t=0..2*Pi );
>
```

In the second section of this worksheet, and also in the last paragraph, we said that a parametric curve in the plane is defined by a single function, a 2-dimensional vector valued function of a single variable. Here is a way to use a Maple function to emphasize that a parametric curve is really defined by a *single* (vector valued) function. The function

$$f(t) = (t\cos(t), t\sin(t))$$

defines a spiral curve. Here is this function defined as a (vector valued) Maple function.

```
> f := t -> (t*cos(t), t*sin(t));
```

Here is how we can use the function **f** with the **plot** command to graph a spiral.

```
> plot( [ f(t), t=0..4*Pi ] );
```

This example shows that a parametric curve is really defined by a single function. When we graph a parametric curve using expressions (instead of a Maple function), the two expressions that we use are the component functions of the single (vector valued) function that defines the curve. It is usually more convenient to work with two expressions than with a single vector valued function, so for the rest of this section we will use expressions to describe parametric curves.

```
>
```

Exercise: By examining the component functions and by relating them to horizontal and vertical movement in the plane, explain why each of the following parametric curves has the shape that it does.

```
> plot( [ t*cos(t),   t*sin(t),   t=-2*Pi..2*Pi],
  scaling=constrained );
> plot( [ t^2*cos(t), t^2*sin(t), t=-2*Pi..2*Pi],
  scaling=constrained );
> plot( [ t*cos(t),   sin(t),     t=-2*Pi..2*Pi],
  scaling=constrained );
> plot( [ sin(t),     cos(t),     t=0..2*Pi ] );
> plot( [ cos(t),     cos(t),     t=0..2*Pi ] );
> plot( [ 3*cos(t),   2*sin(t),   t=0..2*Pi ] );
> plot( [ 3*cos(t),   2*sin(t),   t=0..2*Pi ],
  scaling=constrained );
> plot( [ cos(2*t),   sin(2*t),   t=0..2*Pi ] );
> plot( [ cos(2*t),   sin(t),     t=0..2*Pi ] );
> plot( [ cos(2*t),   sin(3*t),   t=0..2*Pi ] );
> plot( [ cos(3*t),   sin(2*t),   t=0..2*Pi ] );
>
```

Exercise: Try to explain why the following graph has the shape that it does. (Hint: Look at the graphs of the component functions.)

```
> plot([sin(t+sin(t)), cos(t+cos(t)), t=0..2*Pi],
```

```
   scaling=constrained);
> 
```

Just as a single **plot** command can graph more than one function, a single **plot** command can graph more than one parametric curve. For example, here is how we graph two parametric curves (so we have four real valued functions in two pairs). Notice that each parametric curve has its own range and the ranges can be different.

```
> plot( [ [ 3*cos(t), 1/2*sin(t),    t=0..2*Pi ],
>         [ cos(t)*sin(3*t), abs(t), t=-Pi..Pi ] ],
>       color=[blue, red], axes=none );
> 
```

Exercise: Try graphing a real valued function and a parametric curve (i.e., a vector valued function) together in a single **plot** command.

```
> 
```

```
> 
```

5.4.2. Animating parametric curves

When we draw a parametric graph of a function, only the output values of the function get plotted. We lose a lot of information about the function since we do not plot any information about the input values associated with the output values. For example, the following two parametric graphs both draw a circle, but they do not "start" at the same place when $t = 0$, and they do not trace out the circle in the same direction as t increases.

```
> plot( [ cos(t), sin(t), t=0..2*Pi ] );
> plot( [ sin(t), cos(t), t=0..2*Pi ] );
```

As another example, the following parametric graph actually draws the circle three times, but this is not apparent just from the graph since information about the parameter t is not explicitly in the graph.

```
> plot( [ sin(t), cos(t), t=-2*Pi..4*Pi ] );
> 
```

Here is a way to put some information about the parameter values into a parametric graph. Maple has a command, **animatecurve** (from the **plots** package), which can animate the graph of a parametric curve. These animations give us a visualization of how the parameterization traces out the parametric curve. After executing the following **animatecurve** commands, click on the graph, which will initially appear to be empty, and then click on the VCR style "play" button that appears in the animation context bar at the top of the Maple window. Notice how the following two animations show us exactly how these two parameterizations of the circle differ.

```
> plots[animatecurve]( [ cos(t), sin(t), t=0..2*Pi ] );
> plots[animatecurve]( [ sin(t), cos(t), t=0..2*Pi ] );
> 
```

Exercise: Compare the following two animations. What does the second one tell us about the second parameterization as compared to the first parameterization? What information is still not so obvious about the second parameterization even in the second animation?

```
> plots[animatecurve]( [ cos(t),   sin(t),   t=0..2*Pi ] );
> plots[animatecurve]( [ cos(2*t), sin(2*t), t=0..2*Pi ] );
>
```

Exercise: Use the **animatecurve** command to go back and observe the dynamic behavior of the parametric curves from the first two exercises in subsection 5.4.1.

```
>
```

The **animatecurve** command can animate the drawing of more than one parametric curve at a time. To animate more than one parametrice curve, we put a list of parameterizations inside of a pair of braces. Here is an example with an animation related to the second to last exercise.

```
> plots[animatecurve]( { [cos(t), sin(t), t=0..2*Pi],
>                        [.9*cos(2*t), .9*sin(2*t), t=0..2*Pi]
  } );
>
```

Here is another way to animate more than one parameterization at a time. The following group of commands puts three animations into an array variable and then uses the **display** command to display the contents of the array variable. This draws the three animations side by side, as opposed to drawing all three animations on the same pair of axes which is what we would get if we put all three parameterization inside of a single **animatecurve** command.

```
> curves := array( 1..3 ):
> curves[1] := plots[animatecurve]( [cos(t), sin(t),
  t=0..2*Pi],
>                                   scaling=constrained,
  frames=50 ):
> curves[2] := plots[animatecurve]( [sin(t), cos(t),
  t=0..2*Pi],
>                                   scaling=constrained,
  frames=50 ):
> curves[3] := plots[animatecurve]( [cos(2*t), sin(2*t),
  t=0..2*Pi ],
>                                   scaling=constrained,
  frames=50 ):
> plots[display]( curves );
>
```

Here is a way to use **animatecurve** to create animations of parametric curves that are even

more informative than the ones we have created so far. The following example is similar to the previous one but in this example we combine an animation of a parametric curve with animations of its two component functions.

```
> curves := array( 1..3 ):
> curves[1] := plots[animatecurve]( cos(t), t=0..2*Pi,
  frames=50 ):
> curves[2] := plots[animatecurve]( sin(t), t=0..2*Pi,
  frames=50 ):
> curves[3] := plots[animatecurve]( [ cos(t), sin(t), t=0..2*Pi
  ],
>                                   scaling=constrained,
  frames=50,
>                                   color=blue ):
> plots[display]( curves );
```

The two animations on the left are the component functions of the parametric curve animated on the right. Follow the three animations carefully to see how the two component functions correlate with the parametric curve. It may be easier to follow the combined animations if you slow them down or even "single step" through the frames (using the third button from the left in the animation context bar). The animation may look better if you click on the graph and use the mouse to enlarge the graph as much as possible (using the corners of the graph, much like you would enlarge any other window). You can also right click on the graph and use the (pop up) context sensitive plot menu to make the animations have unconstrained scaling (by using the "Projection" item in the pop up menu). That makes the graphs grow in size, but it distorts their appearance a bit.

```
>
```

Here is another way to lay out the above animation. In the following layout, the upper left hand graph describes the horizontal motion (the *x* component) of the parameterization, the lower right hand graph describes the vertical motion (the *y* component) of the parameterization, and the lower left hand graph is the parametric curve. In the upper left hand graph, the time axis is the vertical axis, so the *x*-axis in this graph is parallel to the *x*-axis in the parametric graph (this animation is drawn using a trick from the next subsection for drawing a function with its independent variable on the vertical axis). In the lower right hand graph, the time axis is the horizontal axis, so the *y*-axis in this graph is parallel to the *y*-axis in the parametric graph.

```
> x := t -> cos(t);  # horizontal component function
> y := t -> sin(t);  # vertical component function
> curves := array( 1..2, 1..2 ):
> curves[1,1] := plots[animatecurve]( [ x(t), t, t=0..2*Pi ],
>                                     frames=50 ):
> curves[2,2] := plots[animatecurve]( y(t), t=0..2*Pi,
>                                     frames=50 ):
> curves[2,1] := plots[animatecurve]( [ x(t), y(t), t=0..2*Pi
```

```
  ],
>                                          frames=50, color=blue ):
> curves[1,2] := plot( [[0,0]], axes=none): # create an empty
  graph
> plots[display]( curves );
> x,y := 'x','y':  # unassign x and y
>
```

In the following animation, notice how the horizontal component function makes one sweep from right to left and back again while the vertical component function makes two vertical sweeps from top to bottom. Together, these two motions create a "figure eight" shape. (Try moving the constant 2 from the vertical component function into the horizontal component function. Try replacing the 2 with a 3.)

```
> x := t -> cos(t);  # horizontal component function
> y := t -> sin(2*t);  # vertical component function
> curves := array( 1..2, 1..2 ):
> curves[1,1] := plots[animatecurve]( [ x(t), t, t=0..2*Pi ],
>                                     frames=50 ):
> curves[2,2] := plots[animatecurve]( y(t), t=0..2*Pi,
>                                     frames=50 ):
> curves[2,1] := plots[animatecurve]( [ x(t), y(t), t=0..2*Pi
  ],
>                                     frames=50, color=blue ):
> curves[1,2] := plot( [[0,0]], axes=none): # create an empty
  graph
> plots[display]( curves );
> x,y := 'x','y':  # unassign x and y
>
```

Here is another example of this kind of animation.

```
> x := t -> t*sin(2*Pi*t);  # horizontal component function
> y := t -> t*cos(2*Pi*t);  # vertical component function
> curves := array( 1..2, 1..2 ):
> curves[1,1] := plots[animatecurve]([ x(t), t, t=0..3 ],
>                                    xtickmarks=[],
  ytickmarks=[1,2,3],
>                                    frames=60 ):
> curves[2,2] := plots[animatecurve](y(t), t=0..3,
>                                    xtickmarks=[1,2,3],
  ytickmarks=[],
>                                    frames=60 ):
> curves[2,1] := plots[animatecurve]([ x(t), y(t), t=0..3 ],
>                                    xtickmarks=[],
```

```
   ytickmarks=[],
> frames=60, color=blue ):
> curves[1,2] := plot( [[0,0]], axes=none ): # create an empty
   graph
> plots[display]( curves );
> x,y := 'x','y':  # unassign x and y
>
```

Here is a more complicated example. In this example, the tickmarks on all of the axes have been removed to reduce the clutter in the graphs.

```
> x := t -> sin(t+sin(t));  # horizontal component function
> y := t -> cos(t+cos(t));  # vertical component function
> curves := array( 1..2, 1..2 ):
> curves[1,1] := plots[animatecurve]( [ x(t), t, t=0..2*Pi ],
>                                     xtickmarks=[],
   ytickmarks=[],
>                                     frames=60 ):
> curves[2,2] := plots[animatecurve]( y(t), t=0..2*Pi,
>                                     xtickmarks=[],
   ytickmarks=[],
>                                     frames=60 ):
> curves[2,1] := plots[animatecurve]( [ x(t), y(t), t=0..2*Pi
   ],
>                                     xtickmarks=[],
   ytickmarks=[],
>                                     frames=60, color=blue ):
> curves[1,2] := plot( [[0,0]], axes=none ): # create an empty
   graph
> plots[display]( curves );
> x,y := 'x','y':  # unassign x and y
```

Notice that in these execution groups, the first two lines define the horizontal and vertical component functions. This makes it easy to "plug in" any other parameterization into this animation. Try animating some other parameterizations. (You may or may not want to put tickmarks back into the graphs. Sometimes they make it easier to see what is going on and sometimes they just clutter the graph up too much.)

```
>
```

As we mentioned above, the parameterization $x(t) = \cos(2t)$, $y(t) = \sin(2t)$ for $t \in [0, 2\pi]$ parameterizes two complete "trips" around the circle. But this is not apparent from the graph of the parameterization, which shows just a single circle, and it is not even all that apparent from the **animatecurve** animation of the parameterization, which shows the circle being traced out just one time.

```
> plots[animatecurve]( [cos(2*t), sin(2*t), t=0..2*Pi] );
```

Here is an animation, created using a plot valued function (see Section 10 from Worksheet 4), which clearly shows that this parameterization goes around the circle twice. The **plot** command in the plot valued function graphs a single point with coordinates (cos(2 *t*), sin(2 *t*)) where *t* is the input to the plot valued function. The **seq** command creates a sequence of plots with the input to the plot valued function ranging over the whole domain of our parameterization (i.e., from 0 to 2 π). This gives us a sequence of graphs with the single point travelling along the paramterized curve.

```
> frames := t -> plot( [ [cos(2*t), sin(2*t)] ],
>                      style=point, symbol=diamond):
> seq( frames(i/60*2*Pi), i=0..60 ):
> plots[display]( [%], insequence=true );
>
```

Here is a trick, using the **animate** command, that lets us draw an animation similar to the previous one. In this command, the **t** variable parameterizes just a short segment of the parametric curve. The **s** variable, which is the variable that changes across the frames of the animation, is used to "push" the short segment around the parametric curve. Think of the **s** variable as changing the starting point of the short parameterized segment. As the **s** variable ranges over the interval of our original parametric curve, the short segment travels over the whole curve. Notice that one tricky aspect of this command is that the range of the **t** variable determines just the length of the short segment while the range of the **s** variable determines how far around the circle the segment travels.

```
> plots[animate]( [cos(2*(t+s)), sin(2*(t+s)), t=0..Pi/20],
>                 s=0..2*Pi, frames=60 );
>
```

The next three animations are variations on the last few examples. Try to create other animations that combine techniques from different examples in the section.

```
> frames := s -> plot( [cos(3*Pi*t), sin(Pi*t), t=s..s+1/10] ):
> seq( frames(i/60), i=0..120 ):
> plots[display]( [%], insequence=true );
>
```

```
> background := plot( [cos(3*Pi*t), sin(Pi*t), t=0..2*Pi],
  color=blue ):
> frames := t -> plot( [ [cos(3*Pi*t), sin(Pi*t)] ],
>                      style=point, symbol=diamond):
> seq( frames(i/60), i=0..120 ):
> plots[display]( [%], insequence=true ):
> plots[display]( %, background );
>
```

```
> plots[animate]( [cos(t+s+cos(t+s)), sin(t+s+sin(t+s)),
  t=0..Pi/20],
>                  s=0..2*Pi, frames=60, thickness=2 ):
> plot( [cos(t+cos(t)), sin(t+sin(t)), t=0..2*Pi], color=blue
  ):
> plots[display]( %, %%, axes=none );
>
```

Exercise: The parameterization $x(t) = \cos(2t)$, $y(t) = \sin(3t)$ with $t \in [0, 2\pi]$ traces out the following curve two times.

```
> plot([cos(2*t), sin(3*t), t=0..2*Pi] );
```

Create an animation that verifies this. Also, find a subinterval of $[0, 2\pi]$ such that the parameterization defined on the subinterval traces out the curve exactly one time. Create an animation that verifies this property of your subinterval.

```
>
```

Exercise: Consider the following two parametric curves. In what ways are they similar? In what ways do they differ? What is it about their component functions that cause the similarities and differences?

```
> plot([cos(t), sin(sqrt(2)*t), t=-5*Pi..5*Pi] );
> plot([cos(5*t), sin(7*t), t=-Pi..Pi] );
```

Hint: Analyze the following two parameterizations first.

```
> plot([cos(t), sin(sqrt(2)*t), t=0..5*Pi] );
> plot([cos(5*t), sin(7*t), t=0..Pi] );
>
```

Exercise: Are the following two curves the same?

```
> plot( [cos(7*t), sin(22*t), t=-Pi..Pi],     numpoints=400 );
> plot( [cos(t),   sin(Pi*t), t=-7*Pi..7*Pi], numpoints=400 );
```

Here is a hint. What about the following two?

```
> plot( [cos(7*t), sin(22*t), t=-70*Pi..70*Pi], numpoints=400
  );
> plot( [cos(t),   sin(Pi*t), t=-70*Pi..70*Pi], numpoints=400
  );
>
```

```
>
```

5.4.3. Parametric graphs of real valued functions

Recall that in Section 5.3 on real valued functions we mentioned that the **plot** command, when graphing a real valued function, gives the horizontal axis in Cartesian coordinates the preferred status of being the axis for the independent variable. When graphing parametric equations, the **plot** command does not give either axis a preferred status. It does however always treat the

first expression after the opening bracket as the horizontal component and the second expression as the vertical component. We can use this to get a graph of a real valued function of one variable with the vertical axis as the axis for the independent variable. That is, using the common labels x and y for the horizontal and vertical axes, we can use a parametric curve to draw a graph of $x = f(y)$. For example, here is how we can graph $x = \sin(y)$.

```
> plot( [ sin(y), y, y=0..2*Pi ] );
```

Here is the analogous way to graph $y = \sin(x)$ using a parametric graph.

```
> plot( [ x, sin(x), x=0..2*Pi ] );
```

Here is a graph of a quadratic function with the independent variable along the vertical axis (but notice that we are using the label x for the vertical axis).

```
> plot( [ 3*x^2+5*x-4, x, x=-5..3 ] );
```

The last example shows that the labels x and y really do not mean much to the **plot** command. It is the *order* of the expressions inside of the brackets that determines which axis an expression is used with.

```
>
```

Exercise: Here is a simple piecewise defined function and its usual graph.

```
> f := x -> piecewise( x<1, x, x<2, 1, x<3, cos(2*Pi*x), x<4,
  4-x );
> plot( f, 0..4, scaling=constrained );
```

Use parametric curves to draw each of the following three graphs of **f**.

Part (a) Draw a graph of **f** with the independent variable running along the vertical axis.

Part (b) Draw a graph of **f** with the positive direction of the independent variable running along the left half of the horizontal axis.

Part (c) Draw a graph of **f** with the positive direction of the independent variable running along the bottom half of the vertical axis.

```
>
```

Another application of parametric curves is simultaneously graphing two (or more) real valued functions that have different domains. Recall that in Section 5.3 on real valued functions we mentioned that when the **plot** command is used to graph two or more real valued functions, the functions must all be graphed over the same interval. If we want each function in the graph to have a different range, then we graph the functions separately using multiple **plot** commands and then combine all of the graphs together into a single graph using the **display** command. Another way to accomplish the same effect is to use a single **plot** command and graph each real valued function as a parametric curve, as in the previous paragraph, and give each parametric curve its own range. Here is how we can redo the example, from Section 5.3, of a parabola graphed inside of a semicircle.

```
> fsolve( x^2=sqrt(1-x^2), x, 0..1 ): # compute range for the
  parabola
> plot( [ [t, sqrt(1-t^2), t=-1..1],  # graph the semicircle
>         [t, t^2, t=-%..%]           # graph the parabola
```

```
>        ], color=[green, blue], scaling=constrained
>     );
>
```

Exercise: Use parametric curves to graph the function $x = y^3 - 5y - 1$ and its tangent line at the point $y = 1$. Draw the two curves using a different range for each one.

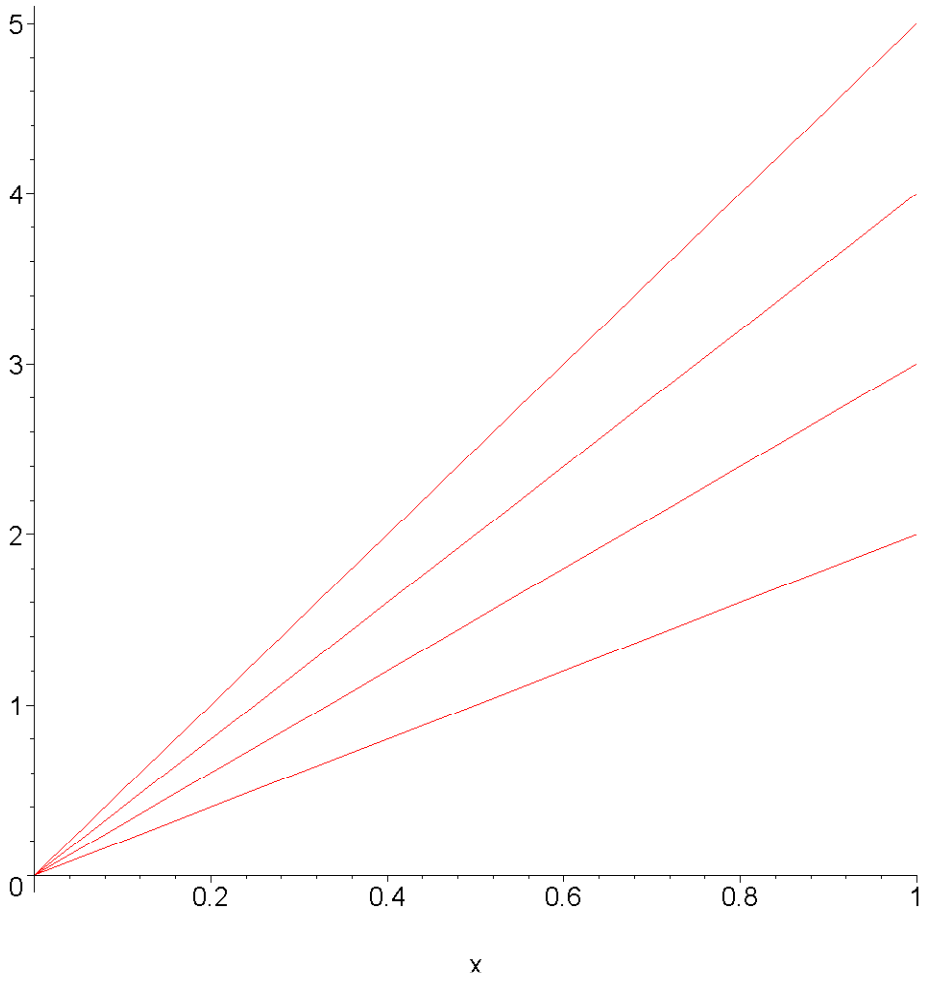
```
>
```

After you have a good graph of the function and its tangent line, try zooming in on the point of tangency.

```
>
```

Exercise: In the following graph, the four line segments all have their right hand endpoint above $x = 1$ and the line segments have different lengths. Redraw the graph, using parametric curves, so that the four line segment all have length 1. Try this two ways, the first way so that all of the line segments have their left hand end at $x = 0$, and another way with all of the right hand endpoints above $x = 1$.

```
> plot( [2*x, 3*x, 4*x, 5*x], x=0..1, color=red );
```




```
>
```

[>

5.4.4. Parametric polygons

The first example of a parametric curve given in almost every calculus book is the standard parameterization of a circle. But that is not really the easiest example to understand. In this subsection we look at several parameterizations that are very similar to the parameterization of a circle, but are easier to derive and understand.

[>

To make things a bit easier as we go along, let us make a small change in the parameterization of the unit circle so that the parameterization goes once around the circle in one unit of time instead of the previous 2π units of time. The new version of the parameterization is given by the following two component functions.

$$x = \cos(2\pi t), \quad y = \sin(2\pi t), \quad t \in [0, 1].$$

The following command graphs the unit circle using this new version of the parameterization.

```
> plot( [cos(2*Pi*t), sin(2*Pi*t), t=0..1] );
```

[>

There is an interesting equation whose graph is a square, centered at the origin, with sides of length two. The equation is

$$\max(|x|, |y|) = 1.$$

Here is its graph.

```
> plots[implicitplot]( max(abs(x),abs(y))=1, x=-1..1, y=-1..1
  );
```

(Notice that the graph has two "clipped" corners. We will see in the next worksheet what causes this anomaly.) What we are going to do is parameterize this curve in a manner similar to the standard parameterization of a crcle.

[>

The basic idea behind parameterizing the square is that we need two functions that "act like" $\cos(2\pi t)$ and $\sin(2\pi t)$ but parameterize the square instead of the circle. Recall that $\cos(2\pi t)$ provides the horizontal component of the motion in the parameterization of the circle and $\sin(2\pi t)$ provides the vertical component of the motion around the circle. Now think about the kind of motion that a parameterization of the square needs to describe. Start the parameterization at time $t = 0$ at the point (1,0) (just like for the circle) and assume that the parameterization will take one unit of time to go around the square (just like for the circle) in the counter clockwise direction and with uniform speed. The horizontal component of the motion around the square should sit still at 1 for a while (how long?), then sweep linearly from 1 to -1 (how long should that take?), then sit still at -1 for a while (again, how long?), then sweep linearly from -1 to 1, and finally, sit still at 1 again for a while. Meanwhile, the vertical component of the motion around the square starts out sweeping linearly up from 0 to 1, then sits still at 1 for a while, then sweeps linearly down from 1 to -1, then holds still at -1 for a while,

and finally sweeps linearly up from −1 to 0.

Read the last paragraph again, and while you are reading it, draw a sketch, on a piece of paper, of the horizontal component function (as a function of time, *t*) that the paragraph describes. Similarly, draw a sketch of the vertical component function (as a function of time, *t*) that the paragraph describes.

```
> 
```

Here is a piecewise defined function that is a first attempt at implementing the horizontal component that we need to parameterize the square.

```
> sq_h := t -> piecewise( t <= 1/8,  1,
>                         t <= 3/8,  1-8*(t-1/8),
>                         t <= 5/8, -1,
>                         t <= 7/8, -1+8*(t-5/8),
>                         t <= 1,    1 );
```

Here is a graph of sq_h(*t*) compared to a graph of cos(2 π *t*).

```
> plot( [ sq_h(t), cos(2*Pi*t) ], t=0..1 );
```

Compare the shape of the graph of sq_h(*t*) to the description given in the last paragraph of the horizontal component of the square's parameterization. Also notice how the graph of sq_h(*t*) is qualitatively very similar to the graph of cos(2 π *t*).

```
> 
```

Now let us define a vertical component function.

```
> sq_v := t -> piecewise( t <= 1/8,  8*t,
>                         t <= 3/8,  1,
>                         t <= 5/8,  1-8*(t-3/8),
>                         t <= 7/8, -1,
>                         t <= 1,    -1+8*(t-7/8) );
```

Here is a graph of sq_v(*t*) compared to a graph of sin(2 π *t*).

```
> plot( [ sq_v(t), sin(2*Pi*t) ], t=0..1 );
> 
```

Here is a graph of our two new component functions together.

```
> plot( [ sq_h(t), sq_v(t) ], t=0..1 );
```

And here is the parametric curve that they define.

```
> plot( [sq_h(t), sq_v(t), t=0..1] );
```

Here is an animation of this parameterization.

```
> plots[animatecurve]( [sq_h(t), sq_v(t), t=0..1], frames=100
  );
> 
```

Here is a graph of a number of "concentric" squares.

```

```

```
> squares := seq( [i*sq_h(t), i*sq_v(t), t=0..1], i=1..10 ):
> plot( [squares] );
>
```

Exercise: Show that for all t between 0 and 1, the point (sq_h(t), sq_v(t)) solves the equation $\max(|x|,|y|) = 1$.

```
>
```

Here is an animation of the circle and square parameterizations together.

```
> plots[animatecurve]( { [sq_h(t), sq_v(t), t=0..1],
>                        [cos(2*Pi*t), sin(2*Pi*t), t=0..1] }
  );
```

(The extra diagonal line at the end of the animation is caused by a bug in **animatecurve**.)

```
>
```

The following animation combines an animation of the parameterization with animations of the horizontal and vertical component functions.

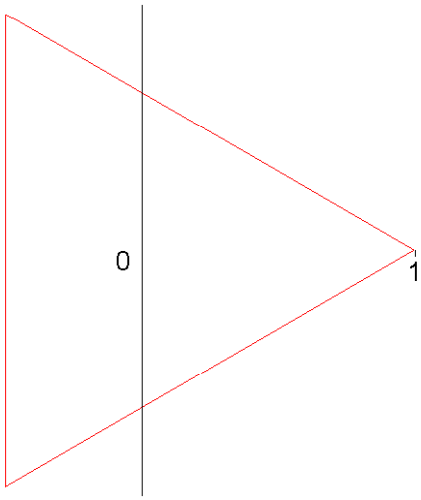
```
> curves := array( 1..2, 1..2 ):
> curves[1,1] := plots[animatecurve]([ sq_h(t), t, t=0..1 ],
>
  xtickmarks=[-1,1],ytickmarks=[1],
>                                    frames=60 ):
> curves[2,2] := plots[animatecurve](sq_v(t), t=0..1,
>                                    xtickmarks=[1],
  ytickmarks=[-1,1],
>                                    frames=60 ):
> curves[2,1] := plots[animatecurve]([ sq_h(t), sq_v(t), t=0..1
  ],
>
  xtickmarks=[-1,1],ytickmarks=[-1,1],
>                                   frames=60, color=blue ):
> curves[1,2] := plot( [[0,0]], axes=none ): # create an empty
  graph
> plots[display]( curves );
>
```

The next animation simultaneously animates the parameterizations of the square and the circle along with animations of their component functions.

```
> curves := array( 1..2, 1..2 ):
> plots[animatecurve]([sq_h(t), t, t=0..1], frames=60,
  color=blue ):
> plots[animatecurve]([cos(2*Pi*t), t, t=0..1], frames=60,
  color=red ):
```

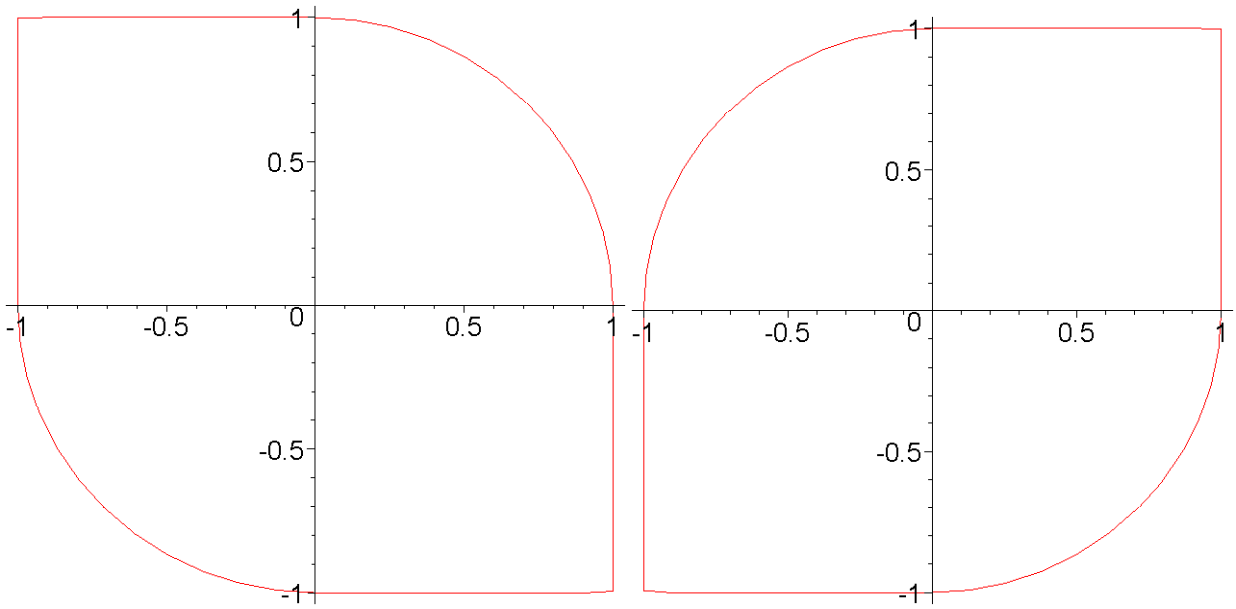
```
> curves[1,1] := plots[display](%%, %):
> plots[animatecurve](sq_v(t), t=0..1, frames=60, color=blue):
> plots[animatecurve](sin(2*Pi*t), t=0..1, frames=60,
  color=red):
> curves[2,2] := plots[display](%%, %):
> plots[animatecurve]([sq_h(t), sq_v(t), t=0..1],
>                     frames=60, color=blue ):
> plots[animatecurve]([cos(2*Pi*t),sin(2*Pi*t),t=0..1],
>                     frames=60, color=red ):
> curves[2,1] := plots[display](%%, %):
> curves[1,2] := plot( [[0,0]], axes=none): # create an empty
  graph
> plots[display]( curves );
>
```

Exercise: Parameterize the equilateral triangle that has its vertices on the circle of radius one centered at the origin and one vertex at the point (1,0).



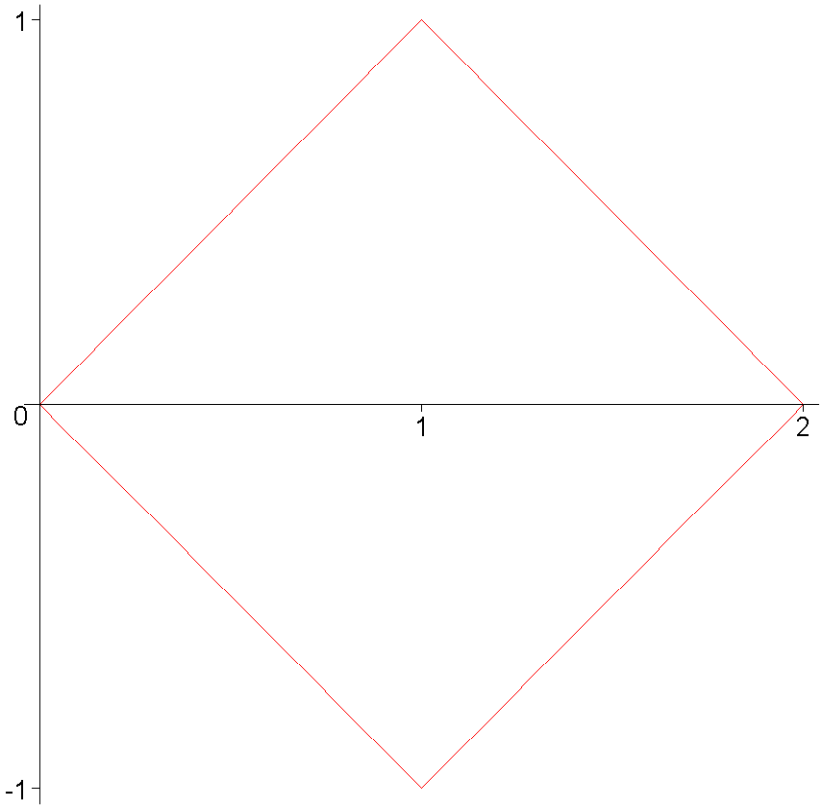

```
>
```

Exercise: Find a parameterization for each of the following two shapes. Each one is half circle and half square.



[>

Let us do another polygon parameterization. Let us parameterize the following square.

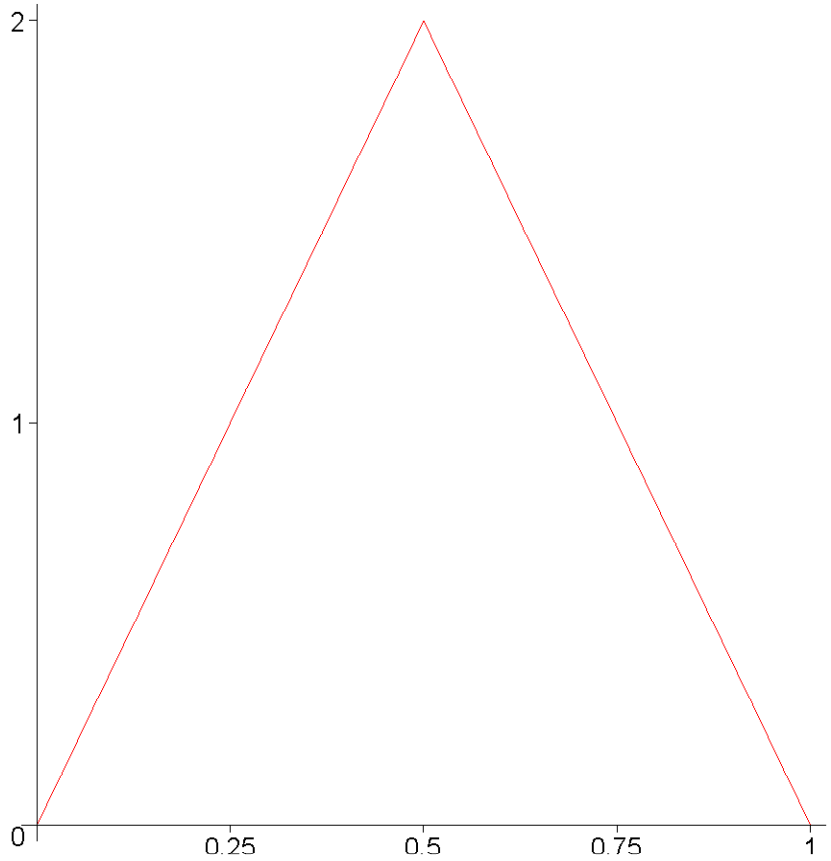


Parameterize it so that we start at the point (0,0), we travel around the polygon in the counter-clockwise direction with uniform speed, and we take one unit of time to make one trip around the polygon.

[

```
> 
```

First, let's think about what this means for the horizontal component function x(t). The horizontal component function should increase from 0 to 1 in 1/4 of a unit of time, and then it should increase from 1 to 2 in another quarter unit of time (which is the same thing as increasing from 0 to 2 in half a unit of time). Then the horizontal component function must decrease from 2 to 0 in the last half unit of time. That describes a piecewise defined function whose graph looks like this (notice that the horizontal axis is time, t, and the vertical axis is the x-axis (why?)).

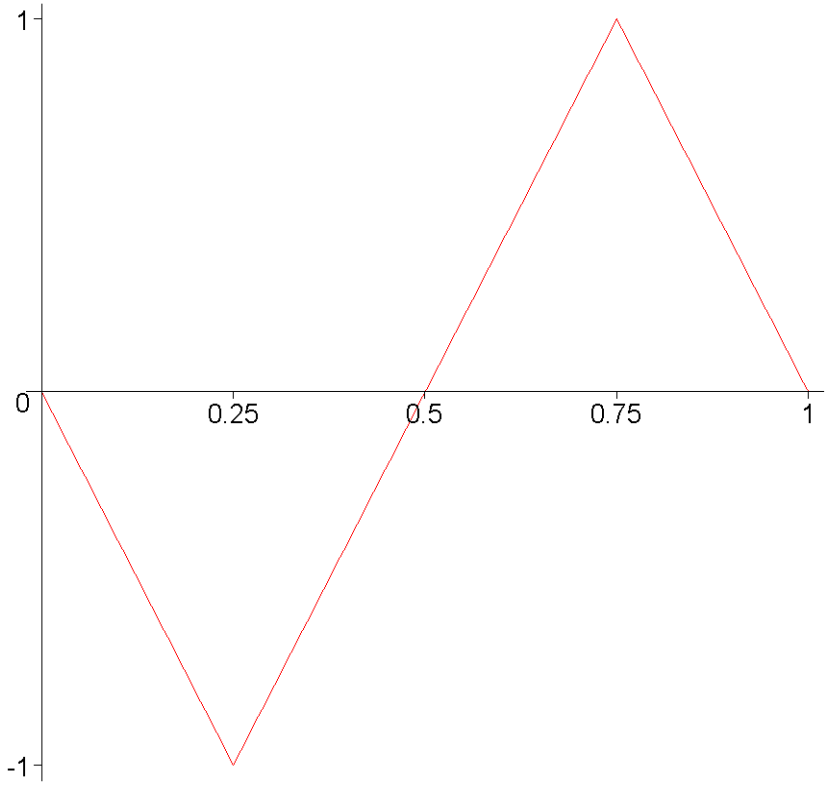


From this graph we can construct the piecewise definition for the horizontal component function. Notice how the second piece of this definition is written in the point-slope form. This makes it easier to figure out the formula from the given graph.

```
> horz := t -> piecewise( t<=1/2,   4*t,
>                         t<=1,   2-4*(t-1/2) ):
> horz(t);
```

Now let us think about the vertical component function y(t). If we look back at the shape that we are trying to parameterize, we see that the vertical component starts at 0 at time 0 and decreases to -1 at time t=1/4 (don't get time, t, mixed up with the horizontal component, x). From time 1/4 to time 1/2, the vertical component should increase from -1 to 0, and then from time 1/2 to time 3/4 the vertical component should increase to 1 (which is the same thing as increasing from -1 to 1 as time increases from 1/4 to 3/4). From time 3/4 to 1, the vertical component should decrease from 1 to 0. All that describes a piecewise defined function whose graph looks like this (and notice that the horizontal axis in this graph is time, t, and the vertical

axis is now the *y*-axis).



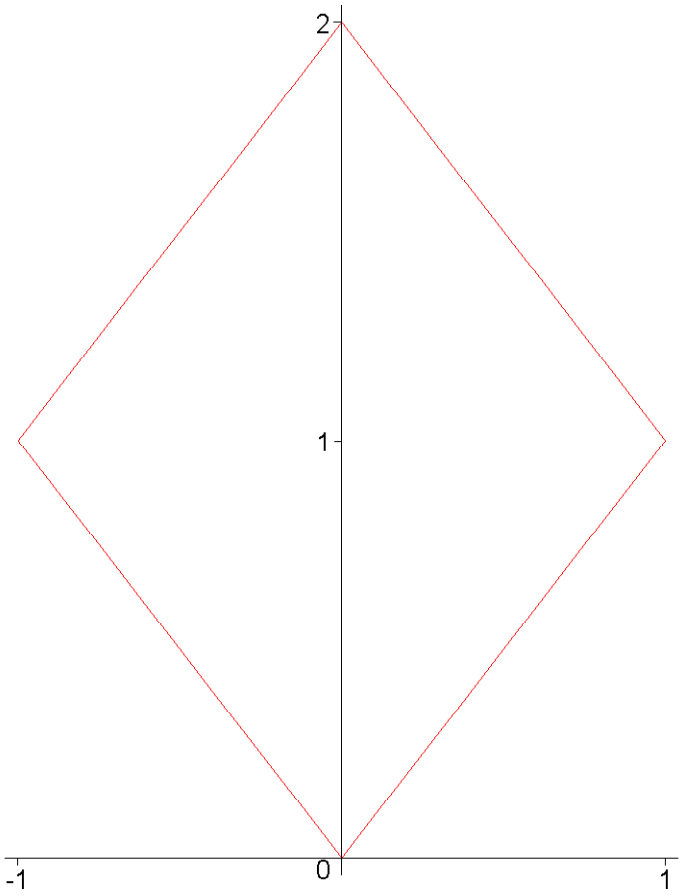
From this graph we can construct the piecewise definition for the vertical component function. Notice how the second and third pieces of this definition are written in the point-slope form. This makes it easier to figure out these formulas from the given graph.

```
> vert := t -> piecewise( t<=1/4,   -4*t,
>                         t<=3/4, -1+4*(t-1/4),
>                         t<=1,    1-4*(t-3/4) ):
> vert(t);
```

Now we can combine the two component functions into a single vector valued function of time. Here is its parametric graph.

```
> plot( [horz(t), vert(t), t=0..1] );
>
```

Exercise: Parameterize the following polygon starting at (0,0), traveling around the polygon in the counter-clockwise direction with uniform speed, and taking one unit of time to make one trip around the polygon.




```
> 
```

What if we come up with trigonometric functions that mimic our two functions **horz** and **vert** ? What kind of shape would those trig functions parameterize? Conside the two functions $2\sin(\pi t)$ and $-\sin(2\pi t)$. Here are their graphs compared with **horz** and **vert**.

```
> plot( [2*sin(Pi*t), -sin(2*Pi*t),
>        horz(t),      vert(t)],    t=0..1,
  color=[blue,blue,red,red] );
```

Here is the shape parameterized by these two trig functions compared with the square.

```
> plot( [ [2*sin(Pi*t), -sin(2*Pi*t), t=0..1],
>         [horz(t),      vert(t),     t=0..1] ],
  color=[blue,red] );
```

Notice how hard it is to predict details about the shape of a parametric curve using just qualitative information about the shape of the component functions.

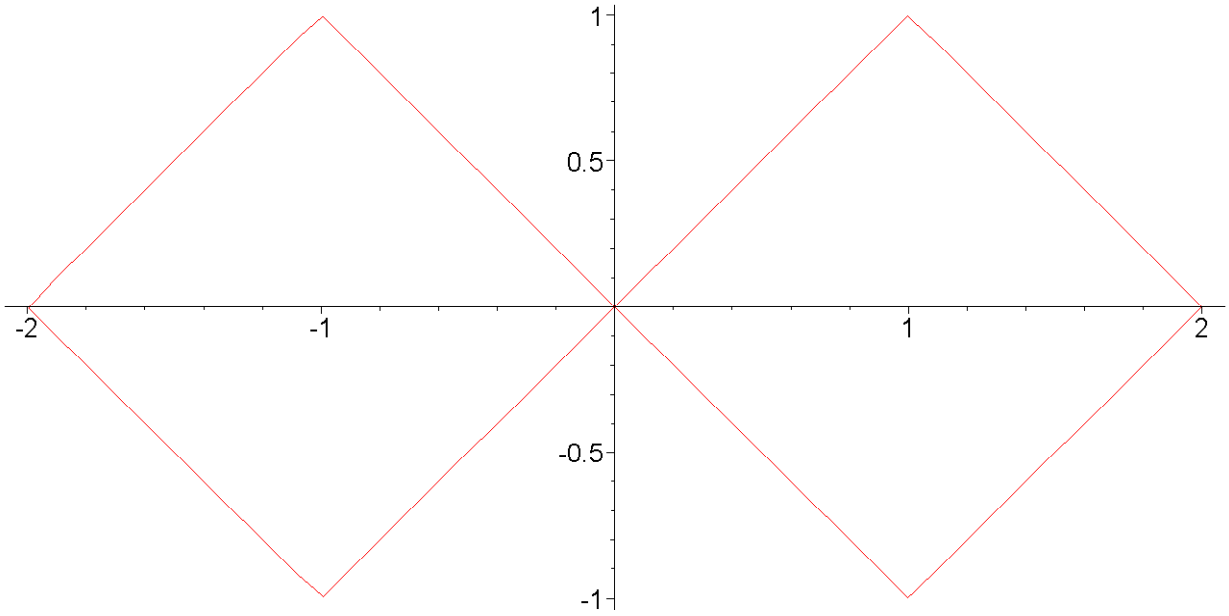
```
> 
```

If we extend the domain for the trig functions, then we see that they in fact parameterize a Lissajous curve in the shape of a figure eight.

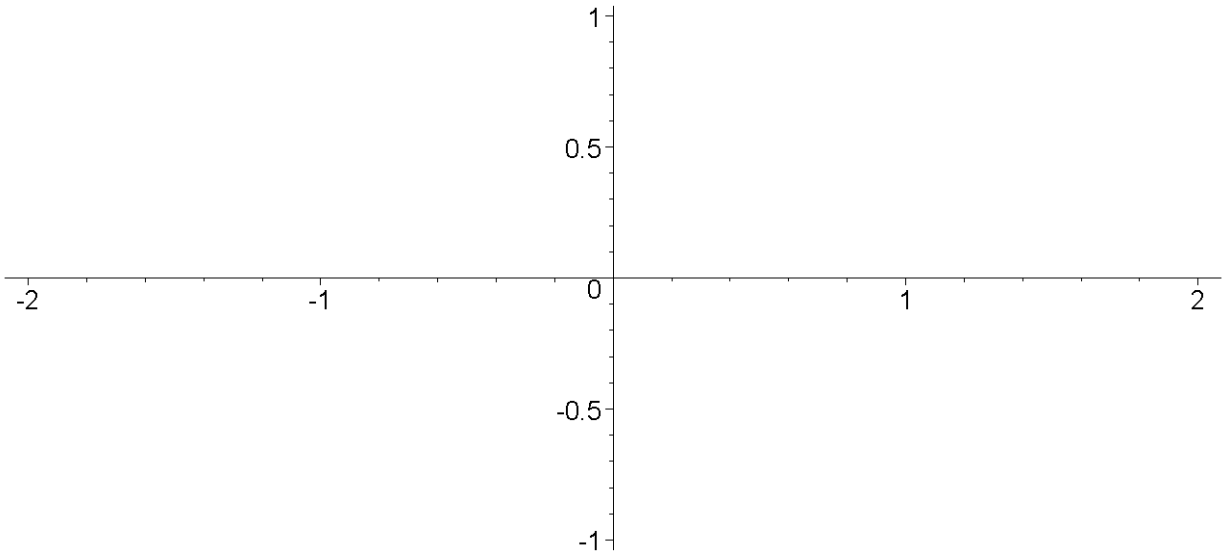
```
> plot( [ [2*sin(Pi*t), -sin(2*Pi*t), t=0..2],
>         [horz(t),      vert(t),     t=0..1] ],
```

```
   color=[blue,red],scaling=constrained);
> 
```

Exercise: Parameterize the following polygon, starting at the point (0,0), traveling around the right half polygon in the counter-clockwise direction with uniform speed, and taking 1/2 unit of time, and then traveling around the left half polygon in the clockwise direction with uniform speed, and taking another 1/2 unit of time.



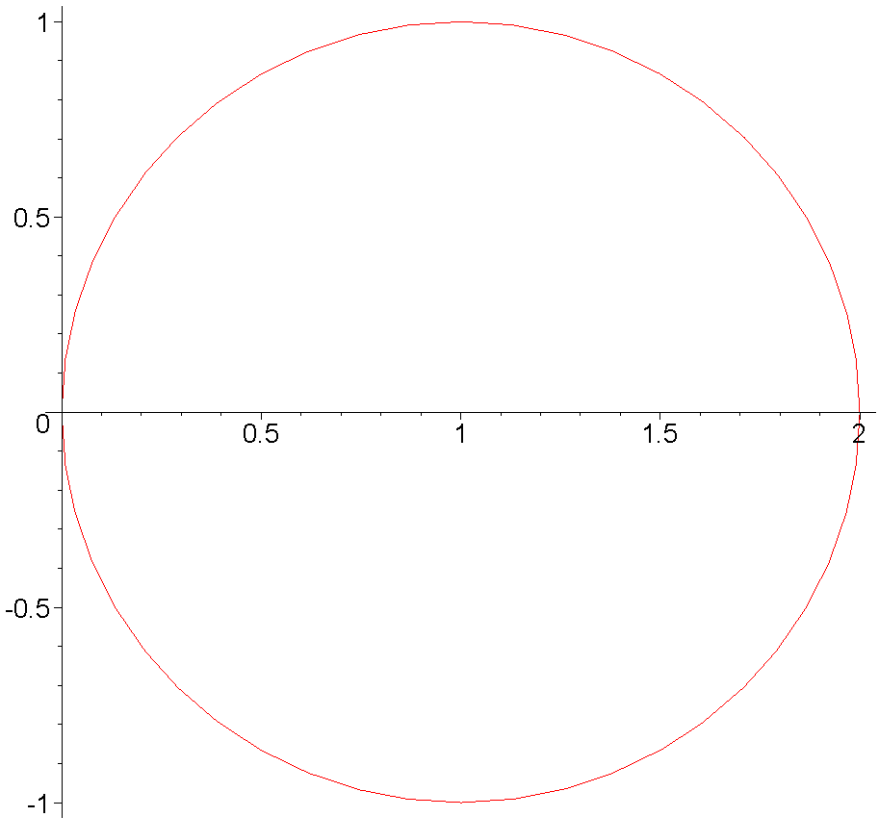
Here is an animation of this parameterization.




```
> 
```

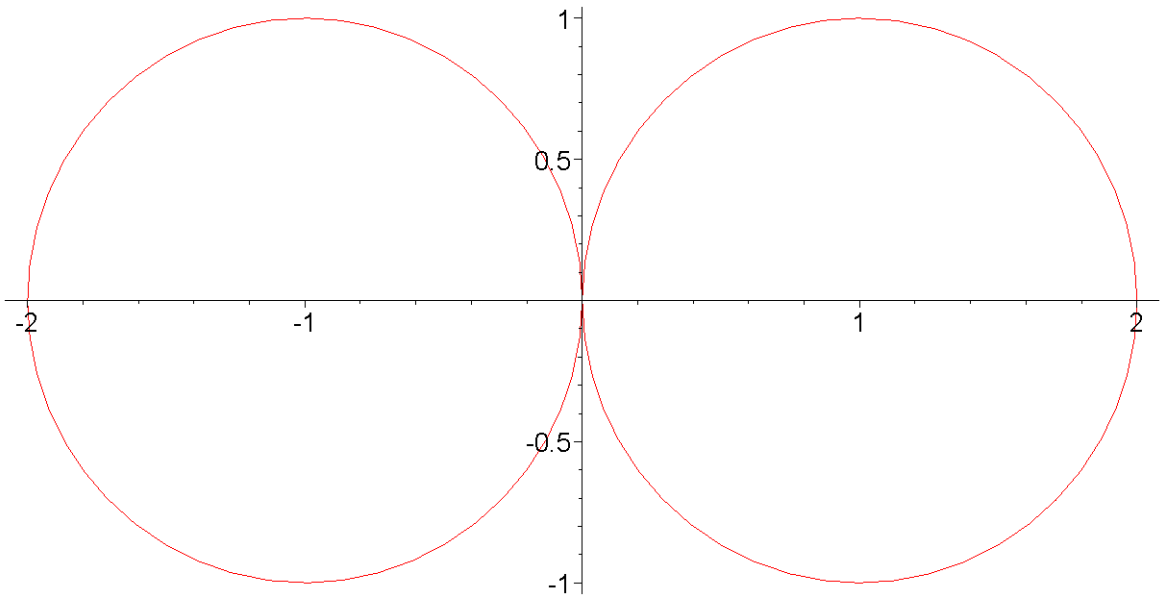
Exercise: Try to come up with another trig function that mimics the shape of **horz**. Use this new function, along with **vert**, to parameterize the following circle, of radius 1, centered at the point (1,0). (Hint: What kind of shape does the new trig function need near $t = 0$ in order to

force the parametric curve to have a vertical tangent line at its origin?)



[>

Exercise: Use a variation on your new function from the previous exercise to parameterize the following figure eight shape made up of two circles. Your parameterization should start at (0,0), go around the right hand circle in the counter-clockwise direction, then go around the left hand circle in the clockwise direction and end at (0,0).

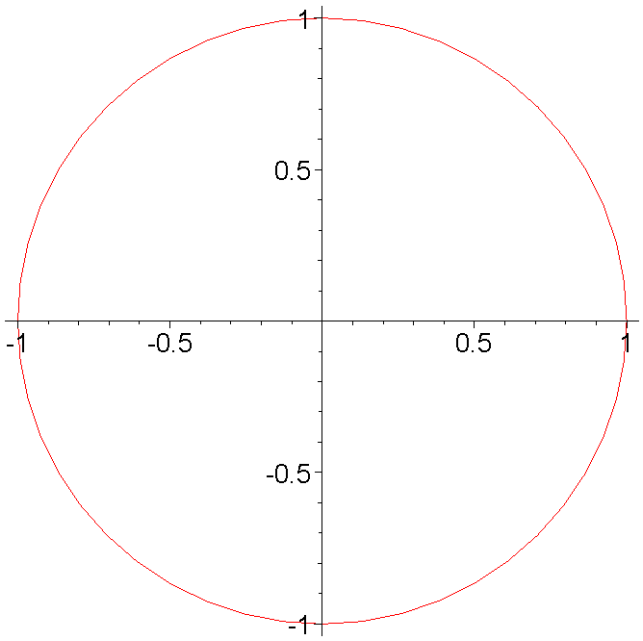


```
>
```

Exercise: Parameterize a regular octagon. (Try to parameterize a regular hexagon also.)

```
>
```

Exercise: In the previous worksheet, in Section 4.10 on plot-valued functions and animations, we showed how to use "homotopies" to create animations that morph the graph of one function into the graph of another function. Pick two parameterizations from this subsection and define homotopies between the horizontal and vertical component functions of the two parameterizations. Use the homotopies to create an animation of one parameterization morphing into the other parameterization. For example, here is an animation of a circle morphing into a triangle.




```
>
```

In the rest of this section we try out some interesting modifications of our first parameterization of the square. Here are the functions that we used to parameterize a square centered at the origin and with side length 2.

```
> sq_h := t -> piecewise( t <= 1/8,  1,
>                         t <= 3/8,  1-8*(t-1/8),
>                         t <= 5/8, -1,
>                         t <= 7/8, -1+8*(t-5/8),
>                         t <= 1,    1 );
> sq_v := t -> piecewise( t <= 1/8,  8*t,
>                         t <= 3/8,  1,
>                         t <= 5/8,  1-8*(t-3/8),
```

```
>                              t <= 7/8, -1,
>                              t <= 1,    -1+8*(t-7/8) );
```

Notice that these two component functions are "flat topped" (which give the parametric graph its straight sides).

```
> plot( [sq_h(t), sq_v(t)], t=0..1 );
```

Here is an interesting modification to **sq_h**. We replace the flat "tops" of **sq_h** with parabolas. This makes the new **sq_h** a bit more cosine like.

```
> new_sq_h := t -> piecewise(t<=1/8,  64*(1-c)*t^2+c,
>                            t<=3/8,  1-8*(t-1/8),
>                            t<=5/8, -64*(1-c)*(t-1/2)^2-c,
>                            t<=7/8, -1+8*(t-5/8),
>                            t<=1,    64*(1-c)*(t-1)^2+c );
```

The following equation was used to help define **new_sq_h**. This equation solves for the coefficients *a*, *b*, and *c* in the general quadratic polynomial $a\,x^2 + b\,x + c$ so that the parabola goes through the points (-1/8, 1) and (1/8, 1).

```
> a, b, c := 'a', 'b', 'c':
> x:=-1/8:
> y:=1/8:
> solve( {a*x^2+b*x+c=1, a*y^2+b*y+c=1}, {a,b,c} );
> x, y := 'x', 'y':
```

The parameter **c** determines the maximum height of the parabolic segments in the graph of **new_sq_h**. Here is a graph of **new_sq_h** with a specific value for **c**.

```
> c := 5/4;
> plot( new_sq_h, 0..1 );
```

Here is how this new function compares with $\cos(2\,\pi\,t)$.

```
> plot( [new_sq_h(t), cos(2*Pi*t)], t=0..1 );
```

Here is the equivalent vertical component function.

```
> new_sq_v := t -> piecewise(t<=1/8,  8*t,
>                            t<=3/8,  64*(1-c)*(t-1/4)^2+c,
>                            t<=5/8,  1-8*(t-3/8),
>                            t<=7/8, -64*(1-c)*(t-3/4)^2-c,
>                            t<=1,   -1+8*(t-7/8) );
```

Now use **new_sq_h** and **new_sq_v** to graph a parametric curve.

```
> plot( [new_sq_h(t), new_sq_v(t), t=0..1], scaling=constrained
  );
```

Try a few different values for the parameter **c**.

```
> c := 3/2;
> plot( [new_sq_h(t), new_sq_v(t), t=0..1], scaling=constrained
  );
```

```
> c := 1/2;
> plot( [new_sq_h(t), new_sq_v(t)], t=0..1 );
> plot( [new_sq_h(t), new_sq_v(t),  t=0..1],
  scaling=constrained );
```

You should try other values of **c** also.

```
>
```

Let us see how we can draw families of these "squares". First, redefine **new_sq_h** and **new_sq_v** so that each is explicitly a function of **c**.

```
> new_sq_h := (t,c) -> piecewise(t<=1/8,  64*(1-c)*t^2+c,
>                                t<=3/8, -8*t+2,
>                                t<=5/8, -64*(1-c)*(t-1/2)^2-c,
>                                t<=7/8,  8*t-6,
>                                t<=1,    64*(1-c)*(t-1)^2+c );
> new_sq_v := (t,c) -> piecewise(t<=1/8,  8*t,
>                                t<=3/8,  64*(1-c)*(t-1/4)^2+c,
>                                t<=5/8,  1-8*(t-3/8),
>                                t<=7/8, -64*(1-c)*(t-3/4)^2-c,
>                                t<=1,   -1+8*(t-7/8) );
```

Now create a sequence of parameters for the **plot** command, each parameter with a different value of **c**.

```
> shapes := seq(
>               [ new_sq_h(t,1/2+i/6), new_sq_v(t,1/2+i/6),
  t=0..1 ],
>               i=0..6
>             ):
> plot( [shapes], scaling=constrained );
```

When **c** is 1, **new_sq_h** and **new_sq_v** parameterizes a square. When **c** is less than 1, the "squares" bulge inwards toward the center, and when **c** is greater than 1 the "squares" bulge outwards.

```
>
```

Here is another family of these shapes. This family of graphs is created in a slightly different way. Here we use the **display** command, plus we parameterize the color of the curves.

```
> frames := s -> plot( [ new_sq_h(t,1/4+s), new_sq_v(t,1/4+s),
  t=0..1 ],
>                      color=COLOR(RGB,i/20.0,0,0) ):
> shapes := seq( frames(i/20), i=1..20 ):
> plots[display]( [shapes], scaling=constrained );
```

Make this into a movie.

```
> plots[display]( [shapes], insequence=true,
  scaling=constrained );
```

Make the movie periodic.

```
> shapes := shapes, seq( shapes[-i], i=1..nops([shapes]) ):
> plots[display]( [shapes], insequence=true,
  scaling=constrained );
>
```

Exercise: Replace the flat tops of **sq_h** (or, to put it another way, the parabolic tops of **new_sq_h**) with piecewise linear tops. So instead of being flat topped like **sq_h**, or with a curved top like **new_sq_h**, the new function will have a "slanted roof" top.

```
>
```

Exercise: Change the definition of **new_sq_h** so that it has two parameters, one for setting the peak of each of the two parabolic "tops". (Then the new version of **new_sq_h** need no longer be symmetric about 1/2.) Draw some parametric curves using this new version of **new_sq_h**.

```
>
```

Exercise: Replace the linear "sides" of **new_sq_h** with parabolic segments. Then the graph of this new function will be piecewise parabolic and there will be two parameters in the definition of the function (or as many as 6 parameters if you do not want to make the parabolic "sides" symmetric). Draw some parametric families with this new function.

```
>
```

```
>
```

5.4.5. Parametric polygon spirals

In the previous section, we pointed out how the horizontal and vertical component functions for our first parameterization of a square, sq_h(t) and sq_v(t), resemble the sine and cosine functions, cos($2\pi t$) and sin($2\pi t$), that parameterize the unit circle.

```
> sq_h := t -> piecewise( t <= 1/8,  1,
>                         t <= 3/8,  1-8*(t-1/8),
>                         t <= 5/8, -1,
>                         t <= 7/8, -1+8*(t-5/8),
>                         t <= 1,    1 );
> sq_v := t -> piecewise( t <= 1/8,  8*t,
>                         t <= 3/8,  1,
>                         t <= 5/8,  1-8*(t-3/8),
>                         t <= 7/8, -1,
>                         t <= 1,    -1+8*(t-7/8) );
> plot( [sq_h(t), cos(2*Pi*t), sq_v(t), sin(2*Pi*t)], t=0..1 );
```

However, there is an important difference between sq_h(t) and cos($2\pi t$) (and also between sq_v(t) and sin($2\pi t$)). The function cos($2\pi t$) is periodic but sq_h(t) is not. In particular,

cos(2 π t) is a periodic function with period 1.

```
> plot( cos(2*Pi*t), t=0..3 );
```

But the graph of sq_h(t) does not repeat itself when we extend its domain beyond 1.

```
> plot( sq_h(t), t=0..3 );
```

The fact that cos(2 π t) and sin(2 π t) are periodic lets us do some nice variations on the standard parameterization of a circle. For example, we can easily parameterize a spiral,

```
> plot( [ t*cos(2*Pi*t), t*sin(2*Pi*t), t=0..3 ] );
```

or draw a Lissajous curve.

```
> plot( [ cos(3*Pi*t), sin(2*Pi*t), t=0..2 ] );
```

We would like to be able to get a "square spiral" from our paramterization of a square, but it doesn't work yet.

```
> plot( [ t*sq_h(t), t*sq_v(t), t=0..3 ] );
```

In order to get a square spiral, we need to make sq_h(t) and sq_v(t) into periodic functions.

```
>
```

Here is one way that we can redefine sq_h(t) so that it is periodic with period 1. We can make sq_h(t) depend only on the fractional part of its input t by composing **sq_h** with the **frac** function.

```
> sq_h := t -> piecewise( frac(t) < 1/8,  1,
>                         frac(t) < 3/8, -8*frac(t)+2,
>                         frac(t) < 5/8, -1,
>                         frac(t) < 7/8,  8*frac(t)-6,
>                         frac(t) < 1,    1 );
```

(Why were all of the **<=**'s changed to **<**'s?) We can do the same for sq_v(t).

```
> sq_v := t -> piecewise( frac(t) < 1/8,  8*frac(t),
>                         frac(t) < 3/8,  1,
>                         frac(t) < 5/8, -8*frac(t)+4,
>                         frac(t) < 7/8, -1,
>                         frac(t) < 1,    8*frac(t)-8 );
```

Now sq_h(t) and sq_v(t) are periodic functions with period 1.

```
> plot( [ sq_h(t), cos(2*Pi*t) ], t=0..3 );
> plot( [ sq_v(t), sin(2*Pi*t) ], t=0..3 );
```

And we can draw a "square spiral".

```
> plot( [ t*sq_h(t), t*sq_v(t), t=0..3 ] );
```

Here is what the two spirals look like together.

```
> plot( [ [t*sq_h(t),     t*sq_v(t),     t=0..3 ],
>         [t*cos(2*Pi*t), t*sin(2*Pi*t), t=0..3 ] ] );
>
```

Here is an animation of the square spiral along with its two component functions.

```
> curves := array( 1..2, 1..2 ):
> curves[1,1] := plots[animatecurve]([ t*sq_h(t), t, t=0..3 ],
>                                    xtickmarks=[],
```

```
   ytickmarks=[1,2,3],
> frames=60 ):
> curves[2,2] := plots[animatecurve](t*sq_v(t), t=0..3,
> xtickmarks=[1,2,3],
   ytickmarks=[],
> frames=60 ):
> curves[2,1] := plots[animatecurve]([ t*sq_h(t), t*sq_v(t),
   t=0..3 ],
> xtickmarks=[],
   ytickmarks=[],
> frames=60, color=blue,
> numpoints=200 ):
> curves[1,2] := plot( [[0,0]], axes=none ): # create an empty
   graph
> plots[display]( curves );
>
```

Exercise: Draw the "square" analogue of the following section of a Lissajous curve.

```
> plot( [ cos(3*Pi*t), sin(2*Pi*t), t=0..1 ] );
>
```

Exercise: Use your solution to an exercise from the last section to create a parameterization of a "triangular spiral".

```
>
```

There is another important difference between sq_h(t) and cos($2\pi t$). The latter is an even periodic function that is defined for all negative values of t, but sq_h(t) is constantly equal to 1 for all negative t.

```
> plot( [ sq_h(t), cos(2*Pi*t) ], t=-2..2 );
```

The fact that cos($2\pi t$) is even periodic (and sin($2\pi t$) is odd periodic) and defined for all negative values of t allows us to draw another kind of spiral, a logarithmic spiral.

```
> plot( [ exp(t)*cos(2*Pi*t), exp(t)*sin(2*Pi*t), t=-3..2 ] );
```

With sq_h(t) and sq_v(t) defined as they are now, we cannot draw a "square logarithmic spiral".

```
> plot( [ exp(t)*sq_h(t), exp(t)*sq_v(t), t=-3..2 ] );
```

We need to extend the definitions of our two component functions to the negative numbers in an appropriate way.

Here is one way we can fix the definition of sq_h(t) so that it too is an even periodic function. We can compose the definition of sq_h(t) with the absolute value function so that sq_h($-t$) = sq_h(t). We can accomplish this by using either the following short, but somewhat obscure, command

```
> sq_h := unapply( (sq_h@abs)(t), t );
```

or we can carefully write out the new definition for sq_h(t).

```
> sq_h := t -> piecewise( frac(abs(t)) < 1/8,  1,
>                        frac(abs(t)) < 3/8,
  -8*frac(abs(t))+2,
>                        frac(abs(t)) < 5/8, -1,
>                        frac(abs(t)) < 7/8,
  8*frac(abs(t))-6,
>                        frac(abs(t)) < 1,    1 );
```

Now sq_h(t) is an even function of period 1, just like cos($2\pi t$).

```
> plot( [ sq_h(t), cos(2*Pi*t) ], t=-2..2 );
```

So we have a horizontal component function sq_h(t) that is defined for all real numbers.

```
>
```

What about the vertical component function? Rather than define an odd periodic extension of sq_v(t) to the whole real line, let us look at this problem from a different point of view. Notice that in the parameterization of the circle, the vertical motion is really the same as the horizontal motion, the vertical motion is just 1/4 of a time unit "out of phase" with horizontal motion. To put it another way, sin($2\pi t$) is just cos($2\pi t$) shifted to the right by 1/4, or sin($2\pi t$) = cos($2\pi(t - .25)$).

```
> cos(2*Pi*(t-1/4));
> combine( % );
```

So let us use this idea to define our vertical component function for the square. We define our vertical component function by shifting our horizontal component function to the right by 1/4. Here is sq_h($t - .25$) compared with sin($2\pi t$).

```
> plot( [ sq_h(t-1/4), sin(2*Pi*t) ], t=-2..2 );
```

So we really only need one function and we can use it for both our horizontal and vertical component functions. Let us use the name **sq** for this function.

```
> sq := eval( sq_h );
```

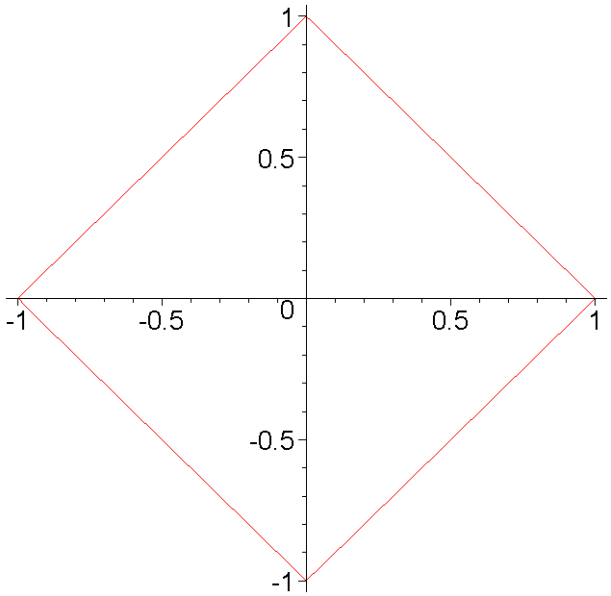
Here is a graph of sq(t) with sq($t - .25$).

```
> plot( [ sq(t), sq(t-1/4) ], t=-3..2 );
```

Here is the parametric "square logarithmic spiral" defined by sq(t) and sq($t - .25$).

```
> plot( [ exp(t)*sq(t), exp(t)*sq(t-1/4), t=-3..2 ] );
> 
```

Exercise: Parameterize the following square figure. Use your parameterization to draw different kinds of "square spirals".



[>

Exercise: Instead of using sq($t - .25$) for the vertical component function, find explicit formulas for the piecewise defined function that defines the odd periodic vertical component function. Call your vertical component function sq2(t) and use it along with sq(t) to draw the square, the square spiral, and the square logarithmic spiral.

[>

Exercise: Start with our original version of sq_h(t) which is only defined for $t \in [0, 1]$.

```
> sq_h := t -> piecewise( t <= 1/8,  1,
>                        t <= 3/8, -8*t+2,
>                        t <= 5/8, -1,
>                        t <= 7/8,  8*t-6,
>                        t <= 1,    1 );
```

Use the **floor** function to modify this version of sq_h(t) and give an alternative definition of the function sq(t) which is the even periodic extension of sq_h(t) to the whole real line. (Hint: Use **floor** to define an expression very similar to **frac** that will work for both positive and negative values of t.)

Then notice that you can use **floor** in the same way in sq_v(t) to extend sq_v(t) to the whole number line as an odd periodic function with period 1.

[>

Exercise: Parameterize a "triangular logarithmic spiral". (Hint: You will need to use the ideas from the previous exercise to get the even and odd extensions of your triangle horizontal and vertical component functions.)

[>

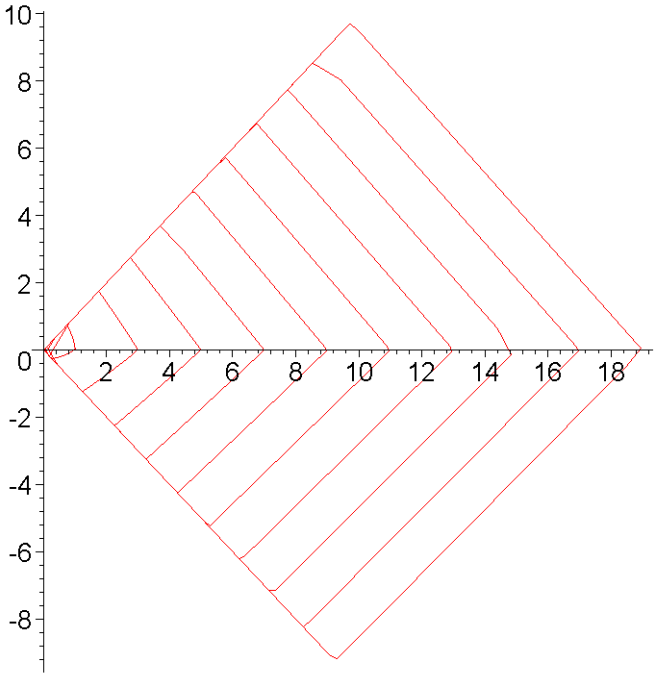
Exercise: Starting with the definitions of **sq_h** and **sq_v** that are periodic, with period 1, but only for the positive half of the number line, here is another way to extend these two functions to be even and odd periodic, respectively, on the whole number line. Here are the definitions of **sq_h** and **sq_v**.

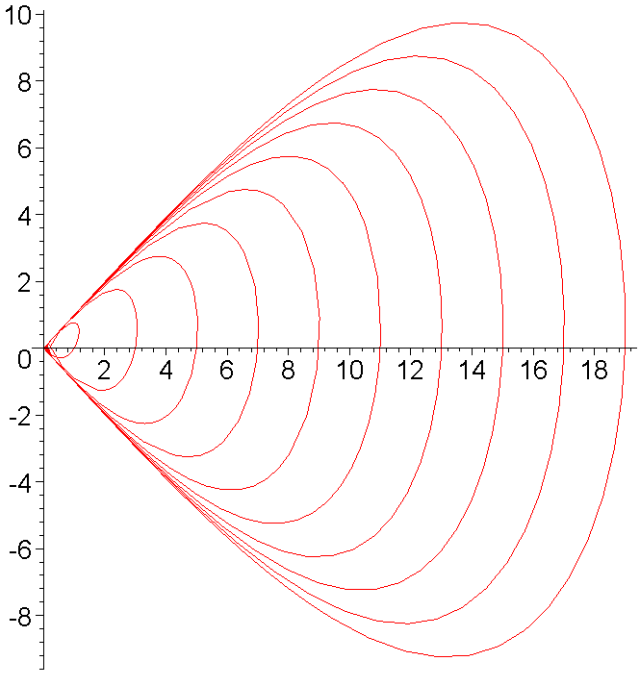
```
> sq_h := t -> piecewise( frac(t) < 1/8,  1,
>                         frac(t) < 3/8, -8*frac(t)+2,
>                         frac(t) < 5/8, -1,
>                         frac(t) < 7/8,  8*frac(t)-6,
>                         frac(t) < 1,    1 );
> sq_v := t -> piecewise( frac(t) < 1/8,  8*frac(t),
>                         frac(t) < 3/8,  1,
>                         frac(t) < 5/8, -8*frac(t)+4,
>                         frac(t) < 7/8, -1,
>                         frac(t) < 1,    8*frac(t)-8 );
> plot( [sq_h, sq_v], -2..2 );
```

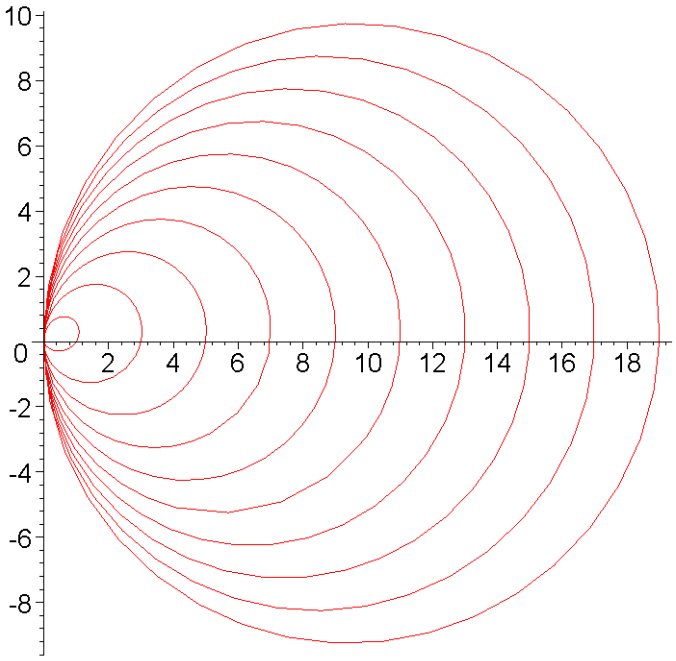
We modify the above two definitions by adding a new line at the beginning of each piecewise definition. Explain why these new definitions work to give even and odd periodic extensions to the whole number line (these new definitions are called **recursive** definitions).

```
> sq_h := t -> piecewise(       t < 0,    sq_h(-t),
>                         frac(t) < 1/8,  1,
>                         frac(t) < 3/8, -8*frac(t)+2,
>                         frac(t) < 5/8, -1,
>                         frac(t) < 7/8,  8*frac(t)-6,
>                         frac(t) < 1,    1 );
> sq_v := t -> piecewise(       t < 0,   -sq_v(-t),
>                         frac(t) < 1/8,  8*frac(t),
>                         frac(t) < 3/8,  1,
>                         frac(t) < 5/8, -8*frac(t)+4,
>                         frac(t) < 7/8, -1,
>                         frac(t) < 1,    8*frac(t)-8 );
> plot( [sq_h, sq_v], -2..2, scaling=constrained );
>
```

Exercise: Modify the "figure eight" parameterizations from the previous section to create the following parametric "spiral like" curves.


10
8
6
4
2
0
-2
-4
-6
-8
2
4
6
8
10
12
14
16
18


10
8
6
4
2
0
-2
-4
-6
-8
2
4
6
8
10
12
14
16
18




```
> 
```

If we return to our modified square parameterization from the previous section, we can create an even periodic extension of the modified horizontal component function.

```
> new_sq := (t,c) -> piecewise(
>    frac(abs(t)) < 1/8,  64*(1-c)*frac(abs(t))^2+c,
>    frac(abs(t)) < 3/8, -8*frac(abs(t))+2,
>    frac(abs(t)) < 5/8, -64*(1-c)*(frac(abs(t))-1/2)^2-c,
>    frac(abs(t)) < 7/8,  8*frac(abs(t))-6,
>    frac(abs(t)) < 1,    64*(1-c)*(frac(abs(t))-1)^2+c
> );
```

Now we can draw very elegant spirals using **new_sq**.

```
> plot( [ t*new_sq(t, 1/2), t*new_sq(t-1/4, 1/2), t=0..4 ],
>       scaling=constrained );
> 
> plot( [ exp(t/2)*new_sq(t, 1/2), exp(t/2)*new_sq(t-1/4, 1/2),
  t=-4..3 ],
>       scaling=constrained );
```

And also some unusual graphs.

```
> plot( [ exp(t/2)*new_sq(t, -1/2), exp(t/2)*new_sq(t-1/4,
  1/2), t=-4..3 ],
>       scaling=constrained );
> 
> plot( [ exp(t/2)*new_sq(t, -1/2), exp(t/2)*new_sq(t-1/4,
  -1/2), t=-4..3 ],
```

```
>       scaling=constrained );
>
```

Exercise: Replace the flat tops of **sq** (or, to put it another way, the parabolic tops of **new_sq**) with piecewise linear tops. So instead of being flat topped like **sq**, or with a curved top like **new_sq**, the new function will have a "slanted roof" top. Use the new function to draw closed parametric curves and spiral parametric curves.

```
>
```

```
>
```

5.4.6. Parametric curves in space: the spacecurve command

A parametric curve in three dimensional space is the output-only graph of a 3-dimensional vector valued function of a real variable. An example of a three dimensional curve is the trefoil knot, which is the parametric graph of the following function.

$$f(t) = \left(\left(2+\cos\left(\frac{3t}{2}\right)\right)\cos(t), \left(2+\cos\left(\frac{3t}{2}\right)\right)\sin(t), \sin\left(\frac{3t}{2}\right)\right)$$

Notice that this function is defined by three real valued (component) functions. If we think of the function $f(t)$ as describing the position of a particle moving in space, then each component function of $f(t)$ describes the particle's motion in the direction of one of the three coordinate axes. In a calculus book, this parameterization might be written as a set of three "parametric equations".

$$x(t) = \left(2+\cos\left(\frac{3t}{2}\right)\right)\cos(t), \quad y(t) = \left(2+\cos\left(\frac{3t}{2}\right)\right)\sin(t), \quad \text{and} \quad z(t) = \sin\left(\frac{3t}{2}\right).$$

Here is one way to use the **spacecurve** command to draw this parametric curve. Be sure to click on the graph and use the mouse to rotate it.

```
> plots[spacecurve]( [ (2+cos(3*t/2))*cos(t),
>                      (2+cos(3*t/2))*sin(t),
>                      sin(3*t/2) ],
>                    t=0..4*Pi
>                  );
```

With the **spacecurve** command, the range for the independent variable can be either inside or outside of the brackets that enclose the three component expressions. In the next example, the range is inside the brackets.

```
> plots[spacecurve]( [ (2+cos(3*t/2))*cos(t),
>                      (2+cos(3*t/2))*sin(t),
>                      sin(3*t/2), t=0..4*Pi ]
>                  );
```

The range for a curve must be inside the brackets if we want to draw more than one curve at a time. Notice how each curve in the next example has its own range, which can be different from the ranges for the other curves.

```
> plots[spacecurve]( { [ (2+cos(3*t/2))*cos(t),
```

```
> (2+cos(3*t/2))*sin(t),
> sin(3*t/2), t=0..4*Pi ],
> [ 5*cos(t), 5*sin(t), 2*cos(6*t),
  t=0..2*Pi ]
> }
> );
```

Notice how, with the **spacecurve** command, multiple space curves are placed inside of a pair of braces, not brackets as in the **plot** command (try changing the pair of braces into a pair of brackets in the last command).

```
>
```

Exercise: Redraw one of the last space curves using a single (vector valued) Maple function to define the parameterization.

```
>
```

Exercise: Suppose we have a 2-dimensional vector valued function of one real variable and suppose that instead of its parametric graph we want its input-output graph. There is an easy way to draw the input-output graph. What is it?
(Hint: You will need to use the **spacecurve** command.)

```
>
```

Let us animate the parameterization of the trefoil knot

$$x(t) = \left(2 + \cos\left(\frac{3t}{2}\right)\right)\cos(t), \qquad y(t) = \left(2 + \cos\left(\frac{3t}{2}\right)\right)\sin(t), \qquad z(t) = \sin\left(\frac{3t}{2}\right).$$

The **animatecurve** command does not work with curves in three dimensional space (and the **animate3d** command only works with surfaces in three dimensions so it also cannot animate a curve in space), so we will do the animation using a plot valued function (see Section 4.10). We will do two animations of the trefoil knot. The first one will have one end of the parameterization fixed and the other end of the parameterization sweeping over the curve until it gets back to the fixed starting point and closes the curve. The second animation will have the two endpoints of the parameterization moving away from the fixed starting point and sweeping out the curve until the two moving points meet and close the curve at a point "opposite" to the fixed starting point.

First, define a plot valued function called **frames**. For any value of this function's input variable, the function returns the graph of a space curve. Notice that in this example, the function being graphed does not depend on the parameter to the plot valued function. Only the range of the graph depends on the parameter (as in **animatecurve**).

```
> frames := s -> plots[spacecurve]( [ (2+cos(3*t/2))*cos(t),
>                                     (2+cos(3*t/2))*sin(t),
>                                     sin(3*t/2), t=0..s ] );
```

Here is an example of evaluating the function **frames**. This evaluation draws half of the trefoil

knot.

```
> frames(2*Pi);
```

Now use the **seq** command and the **frames** function to create a sequence of 50 graphs of segments of the trefoil knot. The **seq** command will evaluate the **frames** function 50 times. Each evaluation of the **frames** function will increment the input of **frames** a little bit and draw a slightly longer piece of the trefoil knot. (Notice the colon at the end of this command so that we do not see the huge amount of data that it creates.)

```
> seq( frames(4*Pi*i/50), i=1..50 ):
```

Now use the **display3d** command with the option **insequence=true** to create an animation out of the 50 frames that we just computed. (Try rotating the animation as it is running to see it from different angles. You can slow it down if you wish, or set it to loop continually.)

```
> plots[display3d]( [%], insequence=true );
>
```

Exercise: Copy the three commands that create the above animation into a single execution group at the end of this exercise. Then modify the example so that the new animation will have the two endpoints of the parameterization moving away from a fixed starting point and sweeping out the curve until the two moving points meet and close the curve at a point "opposite" to the fixed starting point. (This new animation will emphasize the symmetry of the curve.)

```
>
```

```
>
```

5.4.7. The tubeplot command

A nice command in the **plots** package is **tubeplot**, which takes a parametric curve in three dimensional space and graphs a tube around the curve. So, in a sense, this command allows us to convert a one dimensional curve in space into a two dimensional surface. Here is an example of a tube formed around the trefoil knot.

```
> plots[tubeplot]( [(2+cos(3*t/2))*cos(t),
>                   (2+cos(3*t/2))*sin(t),
>                   sin(3*t/2), t=0..4*Pi] );
```

We can change the radius of the tube by using the **radius** option to **tubeplot**.

```
> plots[tubeplot]( [(2+cos(3*t/2))*cos(t),
>                   (2+cos(3*t/2))*sin(t),
>                   sin(3*t/2), t=0..4*Pi], radius=0.4 );
```

In the next sequence of commands, we "cut away" part of the tube and combine it with a graph of the trefoil knot to show how the knot runs through the center of the tube.

```
> f := t -> ((2+cos(3*t/2))*cos(t), (2+cos(3*t/2))*sin(t),
  sin(3*t/2));
> g1 := plots[tubeplot]( [f(t), t=Pi/2..Pi],     radius=0.5 ):
> g2 := plots[tubeplot]( [f(t), t=2*Pi..5*Pi/2], radius=0.5 ):
```

```
> g3 := plots[tubeplot]( [f(t), t=3*Pi..7*Pi/2], radius=0.5 ):
> g4 := plots[spacecurve]( [f(t), t=0..4*Pi] ):
> plots[display]( g1, g2, g3, g4 );
```

Try changing the curve in the last graph into a (very narrow) tube, so that the shape of the curve is more visible. Also, try changing the lengths of the original tubes so that they go around more of the curve.

```
>
```

The radius of the tube can be give by a function, so the radius need not be constant along the whole tube.

```
> plots[tubeplot]( [(2+cos(3*t/2))*cos(t),
>                   (2+cos(3*t/2))*sin(t),
>                   sin(3*t/2), t=0..4*Pi],
  radius=.4+.3*cos(2*t) );
```

Here is another example.

```
> plots[tubeplot]( [t*cos(t), t*sin(t), 0, t=0..7*Pi/2],
>                  radius=t^2/25, scaling=constrained );
```

An easy way to graph a surface of revolution, say of a function f(x) with $x \in [a, b]$, is to use **tupeplot** with the "curve" being the x-axis between a and b and then use the function f(x) as the **radius** option. Here are a couple of examples.

```
> f := x -> x^2;
> a, b := 0, 1:
> plots[tubeplot]( [x,0,0, x=a..b], radius=f(x), axes=normal );
```

```
>
> f := x -> x + sin(x);
> a, b := 0, 5*Pi:
> plots[tubeplot]( [x,0,0, x=a..b], radius=f(x), axes=normal );
>
```

```
>
```

5.5 Non Cartesian coordinate systems in the plane

The **plot** command can use non Cartesian coordinate systems when graphing either real valued functions or parametric curves. The first subsection below reviews the details of polar coordinates, which is the most important and common non Cartesian coordinate system for the plane, and then it briefly introduces a few other non Cartesian coordinate systems. The next two subsections go into graphing real valued and vector valued functions using polar coordinates and a few of the other non Cartesian coordinate systems.

```
>
```

5.5.1. Polar coordinates and other non Cartesian coordinate systems

Recall that for Cartesian coordinates, we start with a special point in the plane, the origin, and then we draw two perpendicular lines through the origin, the coordinate axes. Then every point in the plane is given an "address" made up of two numbers, usually denoted by x and y, where x is the distance of the point from one of the coordinate axes and y is the distance of the point from the other coordinate axis. The coordinate axes are usually, but not always, drawn as a horizontal and a vertical line with the x coordinate being a point's distance from the vertical axis and the y coordinate being a point's distance from the horizontal axis.

In polar coordinates, we again start with a special point in the plane, the origin, and then we draw a single ray (a half line) emanating out of the origin, the **polar axis**. Then every point in the plane is given an "address" made up of two numbers, usually denoted by r and θ, where r measures the point's distance from the origin, and θ measures the point's angle with the polar axis. The polar axis is usually, but not always, drawn as a horizontal ray going to the right of the origin (so the usual polar axis coincides with the usual positive x-axis) with the θ coordinate being a point's counter clockwise angle with the polar axis.

```
> 
```

Here is a simple example. The point in the plane with Cartesian coordinates $(x, y) = (1, 1)$ has polar coordinates $(r, \theta) = \left(\sqrt{2}, \frac{\pi}{4}\right)$. The following two **plot** commands graph this point, first using Cartesian coordinates and then using polar coordinates. Comparing the two graphs, we see that they really do graph the same point.

```
> plot( [ [1,1] ], style=point, symbol=circle );
> plot( [ [sqrt(2),Pi/4] ], coords=polar, style=point,
  symbol=circle );
> 
```

Notice that in the **plot** command, a list of two numbers is used to represent the coordinates of a point in the plane. In the first of the above two **plot** commands, the list **[1,1]** represented the Cartesian coordinates of a point since, by default, the **plot** command uses Cartesian coordinates. In the second **plot** command the list **[sqrt(2),Pi/4]** represented the polar coordinates of a point since we used the **coords=polar** option.

In other situations, Maple uses a different notation to represent the coordinates of a point in the plane. The notation **<a,b>** represents a point in the plane with Cartesian coordinates **a** and **b**. This notation actually represents the point in the plane as a vector by associating the point to the vector whose tail is at the origin and whose tip is at the point. One thing that we can do with this notation is get Maple to convert a point's Cartesian coordinates into polar coordinates and to convert a point's polar coordinates into Cartesian coordinates. For example, here is how we convert the Cartesian coordinates **<1,1>** into polar coordinates.

```
> <1,1>;
> VectorCalculus[MapToBasis]( %, polar );
```

The last result is Maple's way of expressing a point in the plane as a vector in polar coordinates. The last two commands tell us that the Cartesian coordinates $(x, y) = (1, 1)$ are equivalent to the polar coordinates $(r, \theta) = \left(\sqrt{2}, \frac{\pi}{4}\right)$.

```
> 
```

Here is how we convert a pair of polar coordinates into Cartesian coordinates. First, we need to express a point as a vector in polar coordinates. This is a bit cumbersome since the angle bracket notation, **<,>**, defaults to Cartesian coordinates, so we need to tell Maple to give the vector defined with the angle bracket notation the polar coordinate "attribute".

```
> Vector( <sqrt(2),Pi/4>, attributes=[coords=polar] );
```

Now we can calculate the vector's Cartesian coordinates.

```
> VectorCalculus[MapToBasis]( %, cartesian );
```

Notice that Maple choose yet another way to express the point with Cartesian coordinates $(x, y) = (1, 1)$. In this notation, the point (1, 1) is written as a linear combination, with coefficients 1 and 1, of the two standard unit basis vectors for the plane.

```
> 
```

If we know the Cartesian coordinates of a point in the plane, then there are well known formulas for computing the point's polar coordinates from the Cartesian coordinates. Similarly, if we know the polar coordinates of a point in the plane, then there are formulas for computing the point's Cartesian coordinates from the polar coordinates. We can use the **MapToBasis** command to get Maple to tell us these formulas. Here are the formulas for computing the polar coordinates of a point given it Cartesian coordinates.

```
> VectorCalculus[MapToBasis]( <x,y>, polar );
```

In other words, given the Cartesian coordinates (x, y), the polar coordinates are given by $r = \sqrt{x^2 + y^2}$ and $\theta = \arctan\left(\frac{y}{x}\right)$ (look up the **arctan** function in the online help to find out what **arctan(y,x)** means and how it differs from $\arctan\left(\frac{y}{x}\right)$).

Here are the formulas for computing the Cartesian coordinates from the polar coordinates.

```
> Vector( <r,theta>, attributes=[coords=polar] );
> VectorCalculus[MapToBasis]( %, cartesian );
```

In other words, given the polar coordinates (r, θ), the Cartesian coordinates are given by $x = r\cos(\theta)$ and $y = r\sin(\theta)$.

```
> 
```

Exercise: Explain why the following graph shows only one point plotted instead of three.

```
> plot( [ [-sqrt(2),5*Pi/4], [sqrt(2),-7*Pi/4],
  [sqrt(2),9*Pi/4] ],
>       coords=polar, style=point, symbol=circle );
```

```
>
```

Maple can draw "graph paper" for the polar coordinate system. Here is a picture of the polar coordinate grid on the plane.

```
> plots[coordplot]( polar, scaling=constrained );
```

Each circle is made of points that have the same, fixed, r coordinate. Each ray emanating from the origin is made of points that have the same, fixed, θ coordinate. This polar graph paper is analogous to the following Cartesian coordinate grid, in which every vertical line is made of points that have the same fixed x coordinate and every horizontal line is made of points that have the same fixed y coordinate.

```
> plots[coordplot]( cartesian, scaling=constrained, axes=normal
  );
>
```

Polar coordinates are not the only non Cartesian coordinate system that Maple can use on the plane. In fact, Maple knows how to work with 14 non Cartesian coordinates systems for the plane. To give you an idea of what these other coordinates systems can look like, here are a few examples of non Cartesian "graph paper".

```
> plots[coordplot]( bipolar,    scaling=constrained );
> plots[coordplot]( hyperbolic, scaling=constrained );
> plots[coordplot]( rose,       scaling=constrained );
>
```

Given the Cartesian coordinates of a point in the plane, Maple can compute the point's coordinates in any one of the other coordinate systems. Here are the bipolar coordinates of the point with Cartesian coordinates $(x, y) = (1, 1)$.

```
> VectorCalculus[MapToBasis]( <1,1>, bipolar );
```

To see that this result is correct, look at the following graph and notice where the point gets plotted.

```
> plot( [ [arctan(2), ln(5)/2] ], coords=bipolar,
>                                 style=point, symbol=circle );
```

Here are the rose coordinates of the point $(x, y) = (1, 1)$.

```
> VectorCalculus[MapToBasis]( <1,1>, rose );
```

Look back at the graph paper for the rose coordinate system and see if this last result makes sense.

```
>
```

The **MapToBasis** command can give us the coordinate transformation formulas for any of of the non Cartesian coordinate systems. Here are the formulas for computing a point's bipolar coordinates given the point's Cartesian coordinates.

```
> VectorCalculus[MapToBasis]( <x,y>, bipolar );
> convert( %, ln );
```

```
> radsimp( %, ln );
```
And here are the formulas for computing a point's Cartesian coordinates given its bipolar coordinates.
```
> Vector( <u,v>, attributes=[coords=bipolar] ):
> VectorCalculus[MapToBasis]( %, cartesian );
>
```

In fact, we can find the formulas for converting coordinates from any coordinate system into any other coordinate system. Here are the formulas for computing the hyperbolic coordinates of a point given the point's rose coordinates.
```
> Vector( <u,v>, attributes=[coords=rose] ):
> VectorCalculus[MapToBasis]( %, hyperbolic );
> combine( %, symbolic );
> simplify( % ) assuming real;
>
```

The following help page lists all of the two and three dimensional coordinate systems that Maple knows about. For each coordinate system, the transformation formulas are given for converting that coordinate system's coordinates into Cartesian coordinates.
```
> ?coords
```
Here is the graph paper for each of the 15 two dimensional coordinate systems. Because of the following **infolevel** command, each of these **coordplot** commands gives us some information about the graph paper that is not evident just from looking at the graphs. In particular, the information tells us the range that is graphed for each of the coordinate variables in the coordinate system (the output refers to these as the "u range" and "v range" of the graph paper).
```
> infolevel[coordplot] := 2:
> plots[coordplot]( cartesian );
> plots[coordplot]( polar );
> plots[coordplot]( elliptic );
> plots[coordplot]( parabolic );
> plots[coordplot]( hyperbolic );
> plots[coordplot]( cassinian );
> plots[coordplot]( invcassinian );
> plots[coordplot]( rose );
> plots[coordplot]( tangent );
> plots[coordplot]( bipolar );
> plots[coordplot]( cardioid );
> plots[coordplot]( invelliptic );
> plots[coordplot]( logarithmic );
> plots[coordplot]( logcosh );
> plots[coordplot]( maxwell );
>
```

```
>
```

5.5.2. Real valued functions of a single variable in polar coordinates

In polar coordinates, the **plot** command will always graph a real valued function of a real variable as $r = f(\theta)$. That is, the circular direction is the independent variable and the radial direction is the dependent variable.

Here is a well known example of a graph in polar coordinates, a cardiod with $r = 1 + \cos(\theta)$.

```
> plot(1+cos(theta), theta=0..2*Pi, coords=polar,
  scaling=constrained);
>
```

We want to understand this graph and see exactly why it has the shape that it does. To understand graphs in polar coordinates, it helps to think of graphing the function $r = f(\theta)$ on a kind of "radar screen". An angle θ sweeps out, in a counter clockwise direction starting from the horizintal axis, a circle of 2π radians. As θ sweeps around the circle, the function $f(\theta)$ determines the length of a ray emanating from the origin. The tip of this ray, as θ sweeps around the circle, traces out the graph of $r = f(\theta)$. Here is an animation of this. (Do not worry about the code for this animation. It is more important now to understand the relationship between this animation and the graphing of $r = f(\theta)$.)

```
> f := theta -> 1+cos(theta);
> g1 := plots[animatecurve]( [f(theta), theta, theta=0..2*Pi],
>                            coords=polar, frames=60 ):
> p := t -> plot( [r, t, r=0..f(t)], color=black, coords=polar
  ):
> seq( p(2*Pi*n/60), n=0..29 ):  # need to skip t=Pi since
  f(Pi)=0
> seq( p(2*Pi*n/60), n=31..60 ): # and that causes an error in
  plot
> g2 := plots[display]( [%%,%], insequence=true ):
> plots[display]( [g1,g2], view=[-2..2,-2..2],
  scaling=constrained );
>
```

Here is another well known example of a graph in polar coordinates, a spiral with $r = \theta$.

```
> plot(theta, theta=0..4*Pi, coords=polar,
  scaling=constrained);
```

The "radar screen" is swept out two times as θ goes from 0 to 4π, while the radius $r = \theta$ just grows with θ. Here is an animation of this.

```
> f := theta -> theta;
> g1 := plots[animatecurve]( [f(theta), theta, theta=0..4*Pi],
>                            coords=polar, frames=61 ):
```

```
> p := t -> plot( [r, t, r=0..f(t)], color=black, coords=polar
  ):
> plot( [[0,0]], coords=polar ): # empty plot in place of p(0)
> seq( p(4*Pi*n/60), n=1..60 ):
> g2 := plots[display]( [%%,%], insequence=true ):
> plots[display]( [g1,g2], scaling=constrained );
>
```

Another aid in understanding the graph of a polar function $r = f(\theta)$ is to draw the graph of f in rectangular coordinates alongside of the graph of f in polar coordinates. For example, here is the cardiod function $1 + \sin(\theta)$ graphed in both polar and rectangular coordinates.

```
> g := array( 1..2 ):
> g[1] := plot( 1+sin(theta), theta=0..2*Pi, coords=polar ):
> evalf[2]( [Pi/2=``, Pi="Pi", 3*Pi/2=``, 2*Pi="2Pi"] ):
> g[2] := plot( 1+sin(theta), theta=0..2*Pi,
>               xtickmarks=%, ytickmarks=[0,1,2] ):
> plots[display](g, scaling=constrained);
```

Notice that we read these two graphs together by reading the right hand graph from left to right while simultaneously reading the left hand graph counter clockwise starting from the positive horizontal axis. The best way to study the two graphs together is to do it one quadrant at a time, from 0 to $\pi/2$ first, then $\pi/2$ to π, then π to $3\pi/2$, and finally the last quadrant, from $3\pi/2$ to 2π.

```
>
```

Here is an animation of the last graph that combines the "radar screen" animation with a corresponding animation of the rectangular graph.

```
> f := theta -> 1+sin(theta);
> g1 := plots[animatecurve]( [f(theta), theta, theta=0..2*Pi],
>                            coords=polar, frames=60 ):
> p := t -> plot( [r, t, r=0..f(t)], color=black, coords=polar
  ):
> seq( p(2*Pi*n/60), n=0..44 ):  # need to skip t=3*Pi/2 since
  f(Pi)=0
> seq( p(2*Pi*n/60), n=46..60 ): # and that causes an error in
  plot
> g2 := plots[display]( [%%,%], insequence=true ):
> g := array( 1..2 ):
> g[1] := plots[display]( [ g1, g2 ], scaling=constrained ):
> evalf[2]( [Pi/2=``, Pi="Pi", 3*Pi/2=``, 2*Pi="2Pi"] ):
> g1 := plots[animatecurve]( f, 0..2*Pi, frames=60,
  xtickmarks=% ):
> p := x -> plot( [x, y, y=0..f(x)], color=black ):
> seq( p(2*Pi*n/60), n=0..44 ):  # need to skip t=3*Pi/2 since
  f(Pi)=0
> seq( p(2*Pi*n/60), n=46..60 ): # and that causes an error in
```

```
   plot
> g2 := plots[display]( [%%,%], insequence=true ):
> g[2] := plots[display]( [ g1, g2 ], scaling=constrained ):
> plots[display]( g );
```

Notice one again how moving across the right hand graph from left to right corresponds to going around the left hand graph in a counter clockwise direction.

```
>
```

Here is one way to graph a circle in polar coordinates, as the polar graph of $f(\theta) = \sin(\theta)$ with $\theta \in [0, \pi]$.

```
> g := array( 1..2 ):
> g[1] := plot( sin(theta), theta=0..Pi, coords=polar ):
> evalf[2]( [Pi/2="Pi/2", Pi="Pi"] ):
> g[2] := plot( sin, 0..Pi, xtickmarks=%, ytickmarks=[0,1] ):
> plots[display](g, scaling=constrained);
```

Notice how the picture on the right, in rectangular coordinates, shows us how the radius grows from 0 to 1 as θ sweeps from 0 the $\pi/2$ (in the first quadrant), and then the radius shrinks from 1 back to 0 as θ sweeps from $\pi/2$ to π (in the second quadrant).

```
>
```

When graphing a function $r = f(\theta)$ in polar coordinates, it is important to understand how negative values of the radial coordinate r are treated. One of the things that makes interpreting polar graphs ticky is that we do not explicitly notice in the graph how for some angles θ the value of the function $f(\theta)$ may be negative and so the graph of $r = f(\theta)$ is drawn in the quadrant opposite to θ. Here is a graph in which $f(\theta)$ is negative for some θ.

```
> plot( 1+2*cos(theta), theta=0..2*Pi, coords=polar );
```

The best way to tell that the radius r is sometimes negative is to graph the function $f(\theta)$ in Cartesian coordinates.

```
> plot( 1+2*cos(theta), theta=0..2*Pi );
```

Here is how we can use **animate** and **animatecurve** to make more explicit the fact that sometimes the graph of $f(\theta)$ is in the opposite quadrant from θ. This animation looks better if you click on it and then drag one of its corners in order to enlarge the graphs as much as possible.

```
> f := theta -> 1+2*cos(theta);
> g1 := plots[animatecurve]( [f(theta), theta, theta=0..2*Pi],
>                            coords=polar, frames=60 ):
> g2 := plots[animate]( [r, theta, r=0..3], theta=0..2*Pi,
>                       coords=polar, color=black, frames=60 ):
> g := array( 1..2 ):
> g[1] := plots[display]( [ g1, g2 ], scaling=constrained ):
> evalf[2]( [Pi/2=``, Pi="Pi", 3*Pi/2=``, 2*Pi="2Pi"] ):
> g[2] := plots[animatecurve]( f, 0..2*Pi, frames=60,
```

```
   xtickmarks=% ):
> plots[display]( g );
```

The graph on the left is the polar graph of f(θ) and the graph on the right is the cartesian graph of f(θ). In the polar graph, the rotating black ray represents the angle θ. Notice how when the cartesian graph goes negative, the polar graph goes into the quadrant *opposite* to θ. (It might help to view this animation by "single stepping" through the frames, or by slowing the aninmation down.)

```
>
```

Exercise: Earlier we looked at the example of the circle $r = \sin(\theta)$ in polar coordinates with domain $\theta \in [0, \pi]$. Here is the same function with its domain increased to $\theta \in [0, 2\pi]$. Explain why the polar coordinate graph on the left does not seem to change.

```
> g := array( 1..2 ):
> g[1] := plot( sin(theta), theta=0..2*Pi, coords=polar ):
> evalf[2]( [Pi/2=``, Pi="Pi", 3*Pi/2=``, 2*Pi="2Pi"] ):
> g[2] := plot( sin, 0..2*Pi, xtickmarks=%, ytickmarks=[0,1] ):
> plots[display]( g, scaling=constrained );
```

```
>
```

Exercise: Here is an animation, created with a plot valued function, that shows how the family of limacons $f_c(\theta) = 1 + c\sin(\theta)$ depends geometrically on the parameter c.

```
> p := c -> plot( 1+c*sin(theta), theta=0..2*Pi, coords=polar
  ):
> seq( p(n/10), n=-25..25 ):
> plots[display]( [%], insequence=true, scaling=constrained );
```

Add to this animation, alongside of the polar graph of each f_c, the corresponding rectangular graph of f_c. Explain exactly what is is about the family f_c that causes the inner loop to disappear and then reappear.

```
>
```

Note: You can also create the above animation using the **animate** command.

```
> plots[animate]( 1+c*sin(theta), theta=0..2*Pi, c=-2.5..2.5,
>                 coords=polar, scaling=constrained, frames=51
  );
```

```
>
```

Exercise: Do the same thing as in the last exercise but for the family of limacons $g_c(\theta) = c + \sin(\theta)$.

```
> plots[animate]( c+sin(theta), theta=0..2*Pi, c=-2..2,
>                 coords=polar, scaling=constrained, frames=51
  );
```

(Notice that the circle $r = \sin(\theta)$ is a member of this family of limacons.)

```
>
```

Here is another "radar screen" animation of a polar graph alongside of its animated rectangular graph. Why is the range for θ in this graph only from 0 to π?

```
> f := theta -> cos(3*theta);
> g1 := plots[animatecurve]( [ f(theta), theta, theta=0..Pi ],
>          coords=polar, numpoints=200, tickmarks=[2,2],
  frames=60 ):
> g2 := plots[animate]( [ r, theta, r=0..1 ], theta=0..Pi,
>                       coords=polar, color=black, frames=60 ):
> g := array( 1..2 ):
> g[1] := plots[display]( [ g1, g2 ], scaling=constrained ):
> evalf[2]( [Pi/3=``, 2*Pi/3=``, Pi="Pi"] ):
> g[2] := plots[animatecurve]( f(theta), theta=0..Pi,
  frames=60,
>                              xtickmarks=% ):
> plots[display]( g );
```

Try changing the definition of **f** in the above animation.

```
>
```

Exercise: The functions $f(\theta) = \cos(2\theta)$ and $g(\theta) = |\cos(2\theta)|$ seem to have the same graph in polar coordinates.

```
> plot(     cos(2*theta),  theta=0..2*Pi, coords=polar );
> plot( abs(cos(2*theta)), theta=0..2*Pi, coords=polar );
```

However, use animations to show that these two functions trace out their graphs in very different ways as θ varies from 0 to 2 π.

```
>
```

Exercise: Let $f(\theta) = \cos(3\theta)$ and $g(\theta) = |\cos(3\theta)|$ and consider their graphs in polar coordinates. How are the two graphs similar? In what ways do they differ? Be sure to look at animations of how these two functions trace out their graphs as θ varies from 0 to 2 π.

```
> plot(     cos(3*theta),  theta=0..2*Pi, coords=polar );
> plot( abs(cos(3*theta)), theta=0..2*Pi, coords=polar );
>
```

Exercise: Consider the functions $f_n(\theta) = \cos(n\theta)$ in polar coordinates where n is a positive integer and $\theta \in [0, 2\pi]$. For every positive integer n, how many petals does the graph of $f_n(\theta)$ trace out as θ varies from 0 to 2 π?

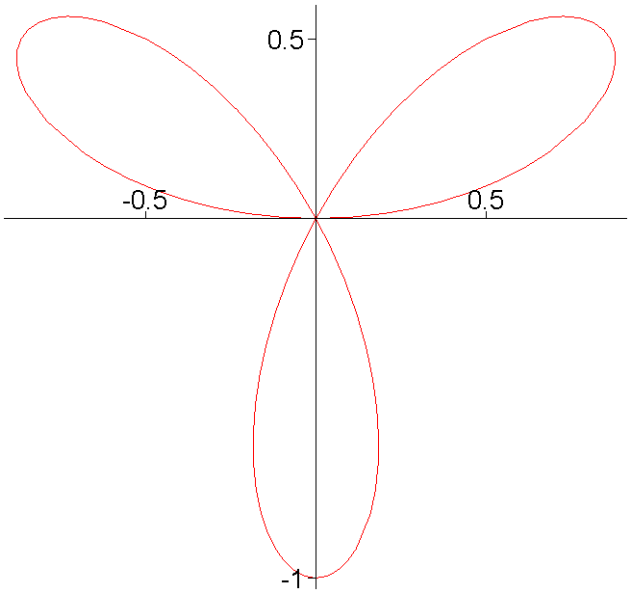
```
>
```

Answer the same question for the functions $g_n(\theta) = |\cos(n\theta)|$.

```
>
```

Exercise: Find a function $r = f(\theta)$ such that its graph in polar coordinates is the following curve

and such that the polar graph traces one complete petal at a time in a counter clockwise direction as θ increases. That is, as θ increases from 0 the polar function should first trace out the complete petal in the first quadrant (going counter clockwise and starting at the origin), then the complete petal in the second quadrant, and then the complete petal that intersects the negative *y* -axis. Use an animation to show that the graph of your polar function has the desired properties. Your animation should show both the polar and the rectangular graphs of your function f.



Exercise: Find a function whose polar graph traces the curve from the last exercise one complete petal at a time in the clockwise direction. That is, the polar function should first trace out the complete petal in the first quadrant (going clockwise and starting at the origin), then the complete petal that intersects the negative *y*-axis, and then the complete petal in the second quadrant. Use an animation to show that the graph of your polar function has the desired properties.

```
> 
```

Here are a number of examples of graphs in polar coordinates of real valued functions of a single variable. These graphs are all examples that are commonly found in calculus books. When you look at these example graphs, it is important to remember that these are input-output graphs of real valued functions in polar coordinates, they are *not* parametric graphs, even though they look a lot like examples of parametric graphs (in Cartesian coordinates). We draw each example using both **plot** and **animatecurve**. The animations show us how each graph develops as the independent variable θ increases, that is, as the angle θ goes around the circle. If you want to see the cartesian graph of these functions, just remove the coords option. (Don't worry about the form of these **animatecurve** commands. They are written this way to avoid a bug in Maple. We will say more about this particular bug below.)

```
> plot( 2*cos(2*theta), theta=0..2*Pi, coords=polar );
> plots[animatecurve]( [2*cos(2*theta), theta, theta=0..2*Pi],
>                      coords=polar, frames=60, numpoints=100
```

```
   );

> plot( cos(x)+sin(x), x=0..Pi, coords=polar );
> plots[animatecurve]( [cos(x)+sin(x), x, x=0..Pi],
>                      coords=polar, frames=60 );

> plot( cos(3*x)+sin(2*x), x=0..Pi, coords=polar );
> plots[animatecurve]( [cos(3*x)+sin(2*x), x, x=0..Pi],
>                      coords=polar, frames=60 );

> plot( 1+4*sin(3*u), u=0..2*Pi, coords=polar );
> plots[animatecurve]( [1+4*sin(3*u), u, u=0..2*Pi],
>                      coords=polar, frames=60, numpoints=200
   );

> plot( y*sin(y), y=-5*Pi..5*Pi, coords=polar,
   scaling=constrained );
> plots[animatecurve]( [y*sin(y), y, y=-5*Pi..5*Pi],
>                      coords=polar, frames=60, numpoints=500,
>                      scaling=constrained, axes=frame );
>
```

Exercise: The above examples all used expressions to represent the functions being graphed. Modify the examples to use Maple functions.

```
>
```

Exercise: Each of the following graphs is drawn as the parametric graph in Cartesian coordinates of a vector valued function. Redraw each of these graphs, as exactly as possible, as the (input-output) graph in polar coordinates of a real valued function.

```
> plot( [ cos(t), sin(t), t=0..2*Pi ] );
> plot( [ t*cos(t), t*sin(t), t=0..4*Pi ] );
> plot( [ 3, t, t=-5..5 ] );
> plot( [ (2+sin(3*t))*cos(t), (2+sin(3*t))*sin(t), t=0..2*Pi ]
   );
>
```

Exercise: The following animation (from the first worksheet) uses parametric equations in Cartesian coordinates. Convert the animation to use the graph in polar coordinates of a real valued function of a single variable.

```
> plots[animate]( [ (1+sin(t)*.5*cos(5*s))*cos(s),
>                   (1+sin(t)*.5*cos(5*s))*sin(s), s=0..2*Pi ],
>                   t=0..2*Pi, scaling=constrained,
   numpoints=100,
```

```
> color=blue, axes=none, frames=60 );
```

Recall that this parametric curve defines a "circle" whose radius (given by the term `1+sin(t)*.5*cos(5*s)`) changes both with angle (the `s` variable) and with time (the `t` variable).

```
>
```

Exercise: The next animation (also from the first worksheet) is based on the previous one. Some of the parameters have been changed and two circles are being morphed simultaneously. Can you convert this animation into one that uses graphs in polar coordinates of two real valued functions of a single variable?

```
> plots[animate]( { [ (1+2*sin(t)*cos(6*s))*cos(s),
>                     (1+2*sin(t)*cos(6*s))*sin(s), s=0..2*Pi
  ],
>                   [ (1+2*sin(t)*cos(6*s))*cos(s),
>                     (1-2*sin(t)*cos(6*s))*sin(s), s=0..2*Pi ]
  },
>                   t=0..2*Pi, scaling=constrained,
  numpoints=150,
>                   color=blue, axes=none, frames=100 );
>
```

Exercise: The following command uses polar coordinates to graph a circle.

```
> plot( sin(t), t=0..Pi, coords=polar, scaling=constrained );
```

If we change the sin to a cos in the last command, we get the circle rotated 90 degrees clockwise.

```
> plot( cos(t), t=0..Pi, coords=polar, scaling=constrained );
```

The following command uses polar coordinates to graph a curve called a cochleoid.

```
> plot( sin(t)/t, t=-6*Pi..6*Pi, coords=polar,
  scaling=constrained );
```

If we change the sin to a cos in the last command, the graph is not rotated 90 degrees.

```
> plot( cos(t)/t, t=-6*Pi..6*Pi, coords=polar,
  scaling=constrained );
```

Find a way to rotate the cochleoid 90 degrees clockwise.

```
>
```

Exercise: The parametric graph in Cartesian coordinates of the vector valued function $f(t) = (3\cos(t), \sin(t))$ is an ellipse.

```
> plot( [ 3*cos(t), sin(t), t=0..2*Pi ], scaling=constrained );
```

Let us try to convert this graph into a graph in polar coordinates of a real valued function. Since $x(t) = 3\cos(t)$, $y(t) = \sin(t)$ and in polar coordinates $r = \sqrt{x^2 + y^2}$, we should graph the function $g(t) = \sqrt{9\cos(t)^2 + \sin(t)^2}$.

```
> plot( sqrt(9*cos(theta)^2+sin(theta)^2), theta=0..2*Pi,
```

```
   coords=polar, scaling=constrained );
```
Why is the graph not an ellipse?
```
>
```

```
>
```

5.5.3. Real valued functions of a single variable in non Cartesian coordinates

Recall that with Cartesian coordinates, if we are given a function, such as sin, we can graph the function two ways, as $y = \sin(x)$ or as $x = \sin(y)$. But in Cartesian coordinates, the **plot** command will only graph the sin function with the independent variable along the horizontal axis, i.e., as $y = \sin(x)$.

```
> plot( sin );
```

So in the Cartesian coordinate system, the **plot** command gives the horizontal axis a preferred status as the default axis for the independent variable of a function.

With polar coordinates, just as with Cartesian coordinates, we can take a function like sin and graph it two ways, as either $r = \sin(\theta)$ or as $\theta = \sin(r)$. But in polar coordinates, the **plot** command will only graph the function sin as $r = \sin(\theta)$.

```
> plot( sin, coords=polar );
```

In the polar coordinate system, the **plot** command uses the circular "direction" as the preferred direction for the independent variable and the radial "direction" is used for the dependent variable. We commonly label the circular direction with θ (the angle) and we label the radial direction with r (the radius). Using these labels, the default way for **plot** to graph a function using polar coordinates is as $r = f(\theta)$.

The labels θ and r may be common for polar coordinates but there is nothing that says that we *must* use them. In fact the **plot** command in polar coordinates does not prefer these labels in any way. Here is a graph of the function $\cos(2u)$ in polar coordinates. Notice that here the **plot** command is using u as the label for the circular direction.

```
> plot( cos(2*u), u=0..Pi, coords=polar, scaling=constrained );
```

```
>
```

When graphing real valued functions of one variable using polar coordinates, the **plot** command will not graph a function with the radial direction as the independent variable. That is, if we use the common labels for the polar coordinates, **plot** will *not* graph $\theta = f(r)$, even though there is nothing to prevent us from defining such a graph. (Later in this section we will use parametric curves in polar coordinates to draw such a graph.)

Here is another example of how **plot** will only graph a real valued function one way, as $r = f(\theta)$. Consider the graph of a constant function. In Cartesian coordinates, the graph of a constant function will be a straight line. The line will be horizontal if the independent variable is

along the horizontal coordinate axis. The graph of a constant function will be a vertical line if the independent variable is along the vertical coordinate axis. The following command tells **plot** to graph several constant functions (in Cartesian coordinates).

```
> plot( [ 1, 3, 6, -4 ] );
```

Since **plot** defaults, in Cartesian coordinates, to having the independent variable along the horizontal axis, we got horizontal lines. Notice that we cannot get **plot** to graph vertical lines (unless we have **plot** draw parametric graphs). In polar coordinates, the graph of a constant function will be either a circle, if the independent variable is in the "circular" direction (i.e., θ), or a ray, if the independent variable is in the "radial" direction (i.e., *r*). The following command tells **plot** to graph several constant functions using polar coordinates.

```
> plot( [ 1, 3, 6, -4 ], coords=polar );
```

Since **plot** defaults, in polar coordinates, to having the independent variable in the circular direction, we got circles. And we cannot get **plot** to graph rays emanating from the origin (unless we use parametric graphs).

```
>
```

The **plot** command can use several different coordinates systems on the plane when it graphs a real valued function of one variable. For example, here is the "graph paper" for a non Cartesian coordinate system called cassinian coordinates.

```
> plots[coordplot]( cassinian );
```

For every one of the non Cartesian coordinates systems, there is a preferred "direction" that (somewhat arbitrarily) is used for the independent variable of the function. Using the discussion from the last paragraph, you should be able to figure out, for each coordinate system, which coordinate direction **plot** defaults to for the independent variable of a function.

Exercise: One of the non Cartesian coordinates systems that **plot** can use on the plane is called hyperbolic coordinates. Here is a picture of some "graph paper" in this coordinate system.

```
> plots[coordplot]( hyperbolic, scaling=constrained );
```

Figure out which of the coordinate directions (the blue or the red one) is the preferred direction that is used as the independent variable when **plot** graphs a real valued function of one variable using the hyperbolic coordinate system.

```
>
```

Exercise: Here are examples of "graph paper" for two more coordinate systems.

```
> infolevel[coordplot] := 2;
> plots[coordplot]( bipolar );
> plots[coordplot]( logcosh );
```

The following help page lists 14 non Cartesian coordinate systems for the plane.

```
> ?plot,coords
```

Draw graph paper for several more of these coordinates systems. Use the online help to read about the **coordplot** command and try to modify some of these pieces of graph paper. Pick a coordinate systems and try to figure out which of its two coordinates is used as the independent

variable by the **plot** command.

```
> 
```

Here is the identuty function, f(t) = t, graphed in seven different coordinate systems in the plane. Notice that this is the function whose graph in cartesian coordinates is the line $y = x$, and in polar coordinates is the spiral $r = \theta$.

```
> plot( t->t, 0..5, coords=cartesian, scaling=constrained );
> plot( t->t, 0..5, coords=polar, scaling=constrained );
> plot( t->t, 0..5, coords=cassinian, scaling=constrained );
> plot( t->t, 0..5, coords=invcassinian, scaling=constrained );
> plot( t->t, 0..5, coords=elliptic, scaling=constrained );
> plot( t->t, 0..5, coords=logcosh, scaling=constrained );
> plot( t->t, 0..5, coords=logarithmic, scaling=constrained );
> 
```

Exercise: Here is one of the above graphs superimposed on its coordinate grid. Determine the value of the independent and dependent variables at the point where this graph has its sharp corner. (Hint: Graph a few constant functions.)

```
> infolevel[coordplot] := 1:
> g1:=plots[coordplot]( cassinian, [0..5,0..2*Pi],
  view=[0..8,0..8] ):
> g2:=plot( t->t, 0..5, coords=cassinian, color=black ):
> plots[display](g1,g2, axes=framed );
> 
```

```
> 
```

5.5.4. Parametric curves in the plane using non Cartesian coordinates

Now let us consider using the **plot** command to draw parametric curves in non Cartesian coordinate systems. The form of the **plot** command for parametric graphs in non Cartesian coordinate systems is the same as it is in Cartesian coordinates. We put the two component functions of the parameterization, along with a range for the independent variable, inside a pair of brackets. But there is one tricky issue here that we need to consider. For each non Cartesian coordinate system, the **plot** command has to make an arbitrary choice of which "direction" of the coordinate system comes first after the opening bracket. In the case of polar coordinates, the first expression after the opening bracket is the radial coordinate and the second expression is the angle. Here is an example that shows this. We graph $r = \cos(2\,\theta)$ using parametric equations (compare this with the graph of the same function in the previous subsection).

```
> plot( [cos(2*t), t, t=0..2*Pi], coords=polar );
```

Notice a subtle difference in how **plot** handles the Cartesian and polar coordinate systems. In the Cartesian coordinate system, the order of the component expressions inside the brackets is independent-variable then dependent-variable (where, by independent and dependent variable,

we mean with respect to how **plot** treats the coordinates when graphing a real valued function instead of parametric equations). So we graph $y = \cos(2x)$ using parametric equations this way.

```
> plot( [x, cos(2*x), x=0..2*Pi] );
```

For the polar coordinate system the, order of the parametric expressions is dependent-variable then independent-variable, the opposite of what it is for Cartesian coordinates.

```
>
```

Exercise: In the next **plot** command, what graph would you expect to get if you removed the option **coords=polar** from the command? Does the graph change from $r = \cos(2t)$ to $y = \cos(2t)$?

```
> plot( [cos(2*t), t, t=0..Pi], coords=polar,
  scaling=constrained );
```

```
>
```

Exercise: The **spacecurve** command can graph parametric curves using cylindrical and spherical coordinates systems (among many others). For each of these two coordinate systems, figure out what the order of the parametric expressions is inside of the brackets.

```
>
```

Knowing how **plot** handles parametric equations in polar coordinates, we can now draw a graph in polar coordinates of a function of the form $\theta = f(r)$. Here is a graph of $\theta = \sin(r)$.

```
> plot( [ r, sin(r), r=0..2*Pi ], coords=polar );
```

```
>
```

Exercise: Study the last graph carefully. Explain why it has the shape that it does. Explain what the following graph is demonstrating.

```
> plot( [ [r,sin(r),r=0..4*Pi], [t,1,t=0..3*Pi],
  [t,-1,t=0..4*Pi] ],
>         coords=polar, color=[red,blue,green] );
```

```
>
```

Exercise: Use polar coordinates to draw a graph of a wedge from a circle.

```
>
```

Exercise: There is a bug in the **animatecurve** command when using polar coordinates. Let us look at an example that brings out this bug. Here is a graph of a function.

```
> plot( sin(4*x), x=0..2*Pi );
```

Let us animate this last graph.

```
> plots[animatecurve]( sin(4*x), x=0..2*Pi );
```

Now let us convert the graph of the function from Cartesian to polar coordinates.

```
> plot( sin(4*x), x=0..2*Pi, coords=polar );
```

Now let us convert the animation from Cartesian to polar coordinates (which should animate the

previous graph).

```
> plots[animatecurve]( sin(4*x), x=0..2*Pi, coords=polar );
```

What went wrong? What did **animatecurve** do? Find a way to use **animatecurve** to animate the correct graph in polar coordinates.

```
>
```

At this point it is worth emphasizing how versatile parametric curves are. Notice how many kinds of graphs we have been able to draw in the last several examples using parametric equations. Besides drawing curves that are not the graph of any function, we have also used parametric curves to draw all four of the kinds of graphs that can be made from a real valued function using Cartesian and polar coordinates. For example, the following four commands use parametric equations to draw graphs of $y = f(x)$, $x = f(y)$, $r = f(\theta)$, and $\theta = f(r)$ respectively, where f is the squaring function.

```
> plot( [ x, x^2, x=-2..2 ] );
> plot( [ y^2, y, y=-2..2 ] );
> plot( [ theta^2, theta, theta=-2..2 ], coords=polar );
> plot( [ r, r^2, r=-2..2 ], coords=polar );
```

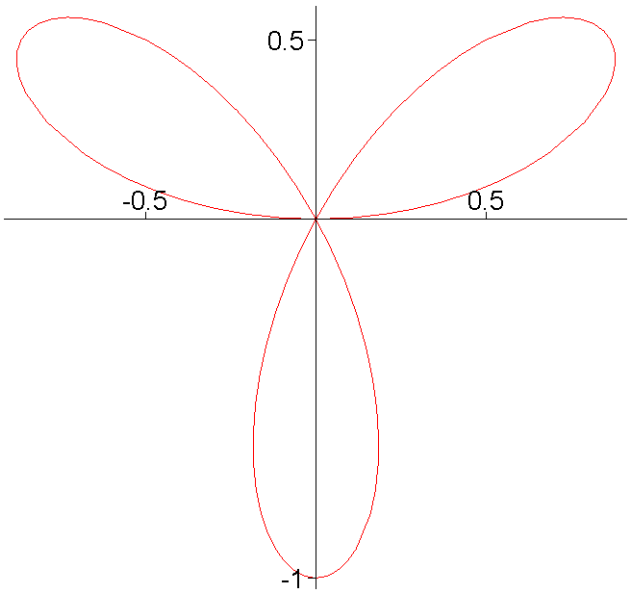
Using parametric equations to graph real valued functions is an important and useful technique. Make sure you understand exactly how the last four examples work. We will return to this idea of using parametric equations to graph real valued functions in the section on parametric surfaces. In that section we will see that this technique lets us work around a couple of bugs in Maple.

```
>
```

Exercise: Try extending the range of the parameter for each of the last two polar graphs. Study these two graphs until they make sense to you.

```
>
```

Exercise: Find a parameterization of the following curve that parameterizes one complete petal at a time in a counter clockwise direction. That is, the parameterization should first trace out the complete petal in the first quadrant (going counter clockwise and starting at the origin), then the complete petal in the second quadrant, and then the complete petal that intersects the negative y-axis. Use an animation to show that your parameterization has the desired properties.



Exercise: Parameterize the curve from the last exercise one complete petal at a time in the clockwise direction. That is, the parameterization should first trace out the complete petal in the first quadrant (going clockwise and starting at the origin), then the complete petal that intersects the negative *y*-axis, and then the complete petal in the second quadrant. Use an animation to show that your parameterization has the desired properties.

[>

[>

5.6. Graphs of real valued functions of two variables

So far in this worksheet we have worked mostly with 2-dimensional graphs. In this and the next two sections we study 3-dimensional graphs. Maple's basic command for graphing in 3-dimensional space is **plot3d**. In this section we see how to use **plot3d** to draw graphs of real valued functions of two variables. In the next section we look at using **plot3d** to draw parametric surfaces (i.e., parametric graphs of 3-dimensional vector valued functions of two real variables). In the section after that we look at how **plot3d** can use non Cartesian coordinate systems in 3-dimensional space.

The following subsection uses **plot3d** to draw graphs of real valued functions of two variables. The next subsection shows how to use **plot3d** to graph functions over regions other than rectangles. Then we look at a way to draw two dimensional representations of three dimensional graphs by using the **contourplot** command. The final subsection shows how to combine **plot3d** with parametric curves in the plane to draw graphs of curves on surfaces.

[>

5.6.1. The plot3d command

If we want to graph a real valued function of two real variables, then its (input-output) graph

will need three dimensions, two dimensions to represent the domain and one more dimension to represent the codomain. We traditionally represent the domain in the graph as a horizontal *xy* -plane and we represent the codomain as a vertical *z*-axis rising up out of the origin of the *xy* -plane. For each ordered pair (x, y) from the domain, we plot in 3-dimensional space the point with Cartesian coordinates (x, y, z) with $z = f(x, y)$ (i.e., we plot the point $(x, y, f(x, y))$). The resulting graph is a 2-dimensional surface in 3-dimensional space. If you like, you can think of the function $f(x, y)$ as a rule that says how to stretch the 2-dimensional *xy*-plane into a 2-dimensional surface in space.

```
> 
```

We use the **plot3d** command to graph a real valued function of two variables by giving **plot3d** the function and two ranges, one range for each of the two independent variables. Here is an example. Be sure to click on this graph with the mouse and try rotating it.

```
> plot3d( cos(2*x)+y^2, x=-3..3, y=-3..3 );
```

When you rotate the graph with the mouse, look at the 3-D graphics context bar at the top of the Maple window. On the left edge of the context bar there are two boxes with numbers in them that change as you rotate the graph. These numbers describe the **orientation** of the graph. If you rotate a graph into a position that you think is especially nice, you can tell the **plot3d** command to draw the graph with that position by specifying the orientation numbers to the **plot3d** command using the **orientation** option. Here is the function from the last graph with an orientation specified in the **plot3d** command.

```
> plot3d( cos(2*x)+y^2, x=-3..3, y=-3..3, orientation=[135,-90]
  );
```

There are many other buttons on the context bar for three dimensional graphs. Try playing with these buttons to see what they do.

```
> 
```

A sometimes useful feature of the **plot3d** command is the ability to graph in "black and white" by using the **shading=zgreyscale** option. Graphs that are drawn this way will often print better on a black and white laser printer than graphs that are drawn in full color.

```
> plot3d( sin(x)+sin(y/2), x=-6..6, y=-6..6, shading=zgreyscale
  );
> 
```

Recall that the **plot** command allowed the specification of two ranges, one range for the independent variable and one range for the dependent variable. The range for the dependent variable was especially useful for graphing functions that had vertical asymptotes. We might therefore expect the **plot3d** command to allow a third range for its dependent variable. But it does not. The **plot3d** command always chooses the range for the dependent variable and it automatically takes care of the fact that a function might "blow up" somewhere. As an example, we graph the function $1/(x\,y)$. Compare graphing this function with graphing $1/x$ using the **plot** command.

```
> plot3d( 1/(x*y), x=-3..3, y=-3..3 );
```
Maple's ability to choose a scale for the vertical axis when there are vertical asymptotes in 3-dimensions is not infallible. The next graph has the same function as the last one, but a slightly different pair of ranges.
```
> plot3d( 1/(x*y), x=-2..2, y=-2..2 );
>
```

The **plot3d** command can draw graphs of several functions at the same time. Here is a graph of two functions. Notice that the two functions *must* be put inside of a pair of braces (brackets would mean something else here).
```
> plot3d( {x*y-1, x^2+y^2}, x=-10..10, y=-10..10 );
>
```

Exercise: Do the two surfaces in the last graph touch each other?
```
>
```

Here is an example of graphing a function and one of its tangent planes. We do this example using Maple functions instead of expressions. First define the function.
```
> f := (x,y) -> -x^2-y^2;
```
Now define the point where we want to compute the tangent plane.
```
> (x0,y0) := (2,2);
```
Now define the function that defines the tangent plane (we use the name **tpf** for "**t**angent **p**lane of **f**").
```
> tpf := (x,y) -> f(x0,y0) + D[1](f)(x0,y0)*(x-x0)
>                          + D[2](f)(x0,y0)*(y-y0);
```
The next command returns the expression for the tangent plane function, just so that we can see what it looks like.
```
> tpf(x,y);
```
Now graph the original function and its tangent plane function.
```
> plot3d( {f, tpf}, -5..5, -5..5 );
```
The last graph used a single **plot3d** command to draw two surfaces. Using a single **plot3d** command to draw multiple surfaces can be convenient, but often we can improve a graph by using separate **plot3d** commands for each surface and then combining the graphs using **display**. Using separate **plot3d** commands lets us, for example, give each surface its own domain. The next execution group improves the last graph by giving the tangent plane a smaller domain.
```
> graph1 := plot3d( f, -5..5, -5..5 ):
> graph2 := plot3d( tpf, -2..5, -2..5 ):
> plots[display]( graph1, graph2 );
>
```

Exercise: Here is the graph of the function and its tangent plane as expressions. For one thing,

notice how much more quickly this graph is drawn.

```
> plot3d( {-x^2-y^2, -8-4*(x-2)-4*(y-2)}, x=-5..5, y=-5..5 );
```

The tangency in this graph is at the point (2,2,-8). Modify the **plot3d** command to "zoom in" on the point of tangency until the function and its tangent plane are just barely distinguishable from each other.

```
>
```

Exercise: Choose some other function and some other point and draw a graph of the function and its tangent plane at the point. Try graphing a function and several tangent planes at once using Maple functions.

```
>
```

Recall from our discussion of the **plot** command that **plot** must, for each coordinate system on the plane, (arbitrarily) choose one coordinate direction for the independent variable of the function. Similarly, the **plot3d** command must, for each coordinate system on three dimensional space, (arbitrarily) choose two coordinate directions for the independent variables of the function. In addition, the **plot3d** command must choose an ordering for the two independent variables, that is, a way to match up each of the two ranges in the **plot3d** command with one of the two preferred coordinate directions.

For example, for the Cartesian coordinate system in space, let us use the common labels *x*, *y*, and *z* for the coordinate axes. The **plot3d** command of course chooses *x* and *y* as the independent variables. The first range in the **plot3d** command is associated to *x* and the second range to *y*. In addition, the *x* and *y* coordinates are drawn on a graph so as to form a right hand coordinate system. So given a function f of two variables, the **plot3d** command will by default draw the graph of $z = f(x, y)$ (in a right hand coordinate system). There are times when some other graph may be desirable, for example $y = f(x, z)$, and we will see in the section on parametric surfaces how this can be done. We will also return to this idea in the section on non Cartesian coordinates in space.

```
>
```

Exercise: The following five commands graph two different functions. Which commands are graphing the same function? Explain why.

```
> plot3d( cos(2*x)+y^2, x=-Pi..Pi, y=-3..3 );
> plot3d( cos(2*x)+y^2, y=-3..3,   x=-Pi..Pi );
> plot3d( cos(2*y)+x^2, y=-3..3,   x=-Pi..Pi );
> plot3d( cos(2*v)+u^2, u=-Pi..Pi, v=-3..3 );
> plot3d( cos(2*u)+v^2, v=-3..3,   u=-Pi..Pi,
  orientation=[135,45] );
>
```

Exercise: Two students are arguing over whether or not, given a function f of two variables,

plot3d can always graph both $z = f(x, y)$ and $z = f(y, x)$ (why would these two graphs be different?). The first student claims it can and gives this example. Let $f(u, v) = u \sin(v)$.

```
> f := (u,v) -> u*sin(v);
```

The next two commands seem to graph $z = f(x, y)$ and $z = f(y, x)$ respectively.

```
> plot3d( f(x,y), x=-2*Pi..2*Pi, y=-2*Pi..2*Pi, axes=framed );
> plot3d( f(y,x), x=-2*Pi..2*Pi, y=-2*Pi..2*Pi, axes=framed );
```

On the other hand, the second student points out that the next command graphs $z = f(x, y)$, and there is no way, using the Maple function form of plotting, to graph $z = f(y, x)$ without redefining f.

```
> plot3d( f, -2*Pi..2*Pi, -2*Pi..2*Pi, axes=framed );
```

Who do you think is more correct? Should we say that **plot3d** can always graph both $z = f(x, y)$ and $z = f(y, x)$, or should we say that **plot3d** graphs only $z = f(x, y)$?

```
>
```

Exercise: If you have read the section on function valued functions from Worksheet 4, then modify the tangent plane example from this subsection so that the function **tpf** is a function valued function of two variables such that the inputs to **tpf** are the coordinates of the point where the tangent plane function for **f** should be computed, and the valued returned by **tpf** is the tangent plane function at that point. So, for example, **tpf(1,2)** would return the function that defines the tangent plane to **f** at the point **(1,2)**. And the expression **tpf(1,2)(2,3)** would evaluate that tangent plane function at the point **(2,3)**, which would give the tangent plane approxiamtion of **f(2,3)**.

Exercise: Further generalize your solution to the last exercise and define a function valued function **tp** with three input variables. The three inputs to **tp** should be a Maple function **f**, and two coordinate variables, **x0** and **y0**, where the tangent plane function for **f** should be computed. The function returned by **tp** should be the function that defines the tangent plane to **f** at the point **(x0,y0)**. So then the expression **tp(f,x0,y0)(x,y)** would be the tangent plane approximation of **f(x,y)** based at **(x0,y0)**.

```
>
```

Compare your solution to the following built in Maple command.

```
> ?VectorCalculus,TangentPlane
>
```

```
>
```

5.6.2. Non rectangular regions

The **plot3d** command can draw graphs over regions that are not rectangular. This can have quite an effect on the appearance of the graph of a function. For example, the following graph of $f(x, y) = x^2 + y^2$ has a rectangular domain.

```
> plot3d( x^2+y^2, x=-4..4, y=-4..4, axes=boxed );
```

Notice how the graph has a very scalloped top edge. If you look at the graph of this function in

most calculus books, the graph will look much more bowl like than the above graph. The following command redraws the graph with a circular, instead of rectangular, domain.

```
> plot3d( x^2+y^2, x=-4..4, y=-sqrt(16-x^2)..sqrt(16-x^2),
  axes=boxed );
```

To see the shape of the region that the function is being graphed over, use the mouse to rotate the graph so that you are looking straight down the *z*-axis onto the *xy*-plane (try this with the last two graphs). Here is the same function graphed over a region that is bounded by a piece of a parabola on one edge and a straight line on another edge.

```
> plot3d( x^2+y^2, x=-4..4, y=-1/2*(x+4)..-(x/2)^2+4,
  axes=boxed );
```

Here is a two dimensional graph of the region that the above graph is drawn over. Try rotating the previous graph so that it looks similar to this next graph.

```
> plot( [-1/2*(x+4), -(x/2)^2+4], x=-4..4,
>       scaling=constrained, color=black );
```

Notice that the region is defined by two functions of *x*. One function defines the "top edge" of the region and the other function defines the "bottom edge". It is also possible to use two functions of *y* to define a region over which a surface is graphed. In this case, one function will define the "left hand edge" and the other function will define the "right hand edge". Here is an example using the same surface as above. The "right hand edge" of the graphing region is a sine curve and the "left hand edge" is a vertical line.

```
> plot3d( x^2+y^2, x=-4..sin(Pi*y)+3, y=-4..4, axes=boxed );
```

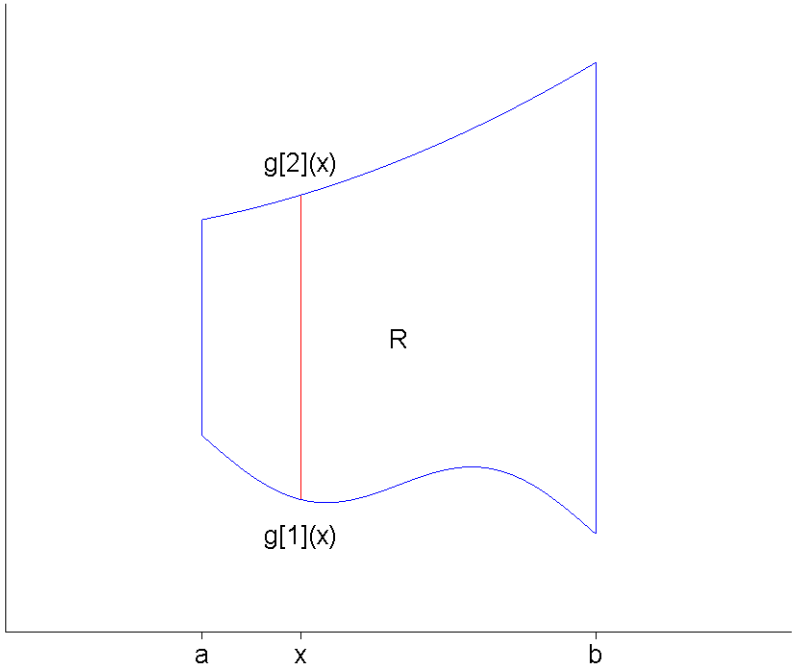
Here is the region drawn by itself in two dimensions (notice that this time we need to use parametric curves to outline the region). Again, line up the previous graph so that it looks similar to the next graph.

```
> plot( [ [-4, y, y=-4..4], [sin(Pi*y)+3, y, y=-4..4],
>         [x, 4, x=-4..3], [x, -4, x=-4..3] ],
>       scaling=constrained, color=black );
>
```

Exercise: The following command draws both a surface in 3-dimensional space and its region in the *xy*-plane. Explain what trick is being used to graph the region in the *xy*-plane.

```
> plot3d( {0,x^2+y^2}, x=-4..sin(Pi*y)+3, y=-4..4, axes=boxed
  );
>
```

Let us go into more detail about graphing function over non-rectangualr regions. The **plot3d** command can graph functions over two kinds of non rectangular regions. We call these two kinds of regions Type I and Type II regions. Here is a general picture of what a Type I region looks like.



A Type I region R is defined by two numbers a and b and two functions $g_1(x)$ and $g_2(x)$ defined on the interval $[a, b]$. The Type I region R is defined as all of the points in the xy-plane that have their x value in the interval $[a, b]$ and have their y value between $g_1(x)$ and $g_2(x)$. That is, the region R is defined as

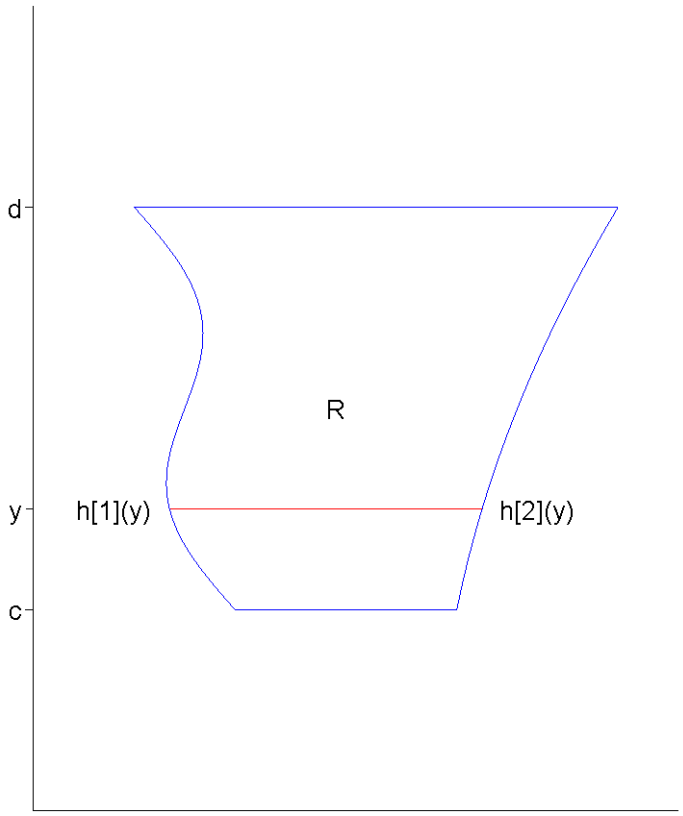
$$R = \{ (x, y) \mid a \le x \text{ and } x \le b, \text{ and } g_1(x) \le y \text{ and } y \le g_2(x) \}.$$

Given a function $f(x, y)$ and a Type I region R, we graph the function over the region by using a **`plot3d`** command of the following form.

`plot3d( f(x,y), x=a..b, y=`g_1`(x)..`g_2`(x) )`

Notice how the y range depends on x. Every choice of x between a and b determines a different range for y. That is, the endpoints for the y range move up and down as x varies from a to b (look again at the above figure).

Here is a general picture of what a Type II region looks like.



A Type II region R is defined by two numbers c and d and two functions h_1 and h_2 defined on the interval $[\,c, d\,]$. The Type II region R is defined as all of the points in the xy-plane that have their y value in the interval $[\,c, d\,]$ and have their x value between $h_1(y)$ and $h_2(y)$. That is, the region R is defined as

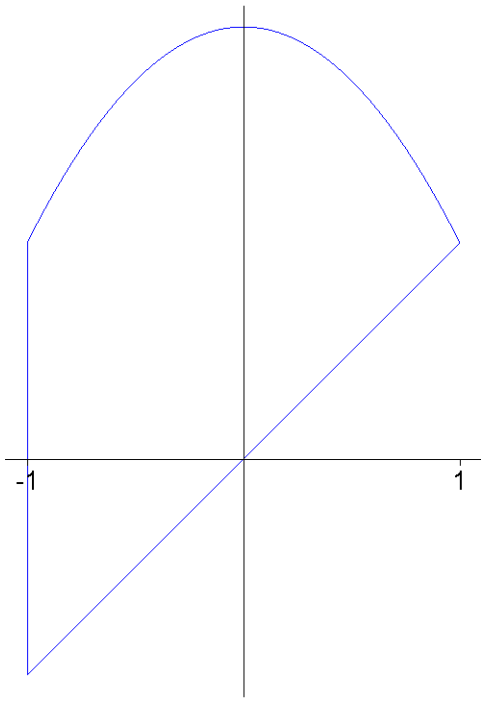
$$R = \{\ (x, y) \mid c \le y \text{ and } y \le d, \text{ and } h_1(y) \le x \text{ and } x \le g_2(y)\ \}.$$

Given a function f(x, y) and a Type II region R, we graph the function over the region by using a **`plot3d`** command of the following form.

`plot3d( f(x,y), x=`h_1`(y)..`h_2`(y), y=c..d )`

Notice how the x range depends on y. Every choice of y between c and d determines a different range for x. That is, the endpoints for the x range move left and right as y varies from c to d (look at the above figure). Notice that in the **`plot3d`** command we still put the x range before the y range, even though the x range depends logically on y.

In a Type I region, the left and right hand "sides" of the region do not have to be vertical lines. They can each reduce to just a point. For example, in the following Type I region, the interval on the x-axis is $[\,a, b\,] = [\,-1, 1\,]$, the top edge is defined by the function $g_2(x) = 2 - x^2$, and the bottom edge is defined by $g_1(x) = x$. The right hand "edge" of this region is just a single point.



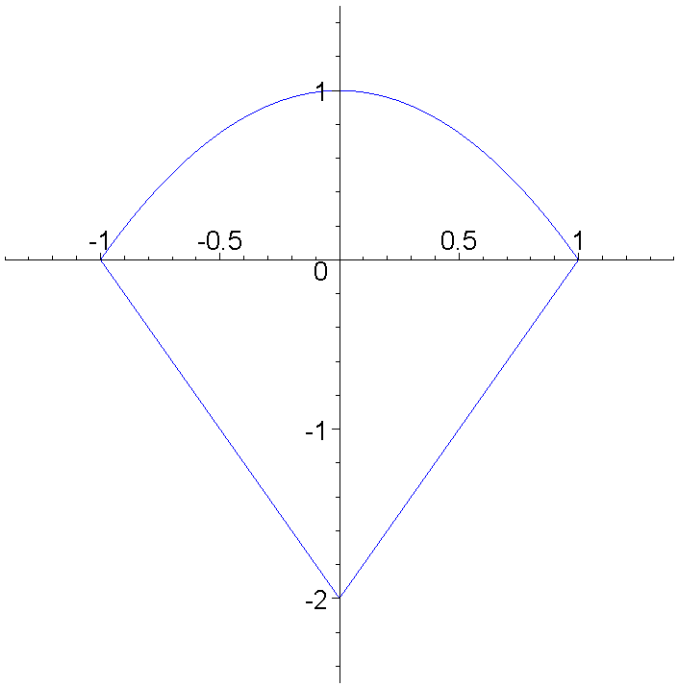
Exercise: Make up an example of a Type II region in which the top and bottom "edges" are just points.

[**>**

It is important to realize that the upper and/or lower functions for a Type I region can be piecewise defined functions. For example, in the Type I region shown below, the interval on the x-axis is $[a, b] = [-1, 1]$, the top edge is defined by the function $g_2(x) = 1 - x^2$, and the bottom edge is defined by the piecewise function

$$g_1(x) = \begin{cases} -2(x+1) & x \le 0 \\ 2(x-1) & 0 < x \end{cases}$$

The left "edge" of this Type I region is the single point (-1,0) and the right "edge" is the point (1,0).



Exercise: The above region is also a Type II region where the interval on the y-axis is $[c, d] = [-2, 1]$, and the left and right hand side functions $h_1(y)$ and $h_2(y)$ are both piecewise defined functions (and the top "edge" is the single point (0,1) and the bottom "edge" is the point (0,-2)). Find the piecewise fomulas for both $h_1(y)$ and $h_2(y)$ and use your formulas to redraw the region.

```
> 
```

Exercise: The following graph of the function $f(x, y) = x^2 - y^2$ has a very curved bottom.

```
> plot3d( x^2-y^2, x=-4..4, y=-4..4, axes=framed );
```

Find a shape for the domain of the graph so that the graph of this function has a flat bottom.

```
> 
```

In a previous exercise, we drew a graph of a function and one of its tangent planes. Here is the graph once again.

```
> plot3d( {-x^2-y^2, -8-4*(x-2)-4*(y-2)}, x=-5..5, y=-5..5 );
```

Let us modify the region that this graph is drawn over so that the graph of the function has a flat bottom. We will draw the graph over a circular domain.

```
> plot3d( {-x^2-y^2, -8-4*(x-2)-4*(y-2)},
>          x=-5..5, y=-sqrt(25-x^2)..sqrt(25-x^2) );
```

This graph is a bit strange looking. The tangent plane is now a very large disk, and it does not look very good. We really should graph the function over a circular domain and graph the tangent plane over a rectangular domain. We can do this if we use two separate **plot3d** commands and then combine their graphs using the **display** command.

```
> graph1 := plot3d( -x^2-y^2, x=-5..5,
```

```
   y=-sqrt(25-x^2)..sqrt(25-x^2) ):
> graph2 := plot3d( -8-4*(x-2)-4*(y-2), x=0..4, y=0..4 ):
> plots[display]( graph1, graph2 );
>
```

Exercise: We used expressions in the last example. Modify the example to use Maple functions throughout, including using functions to define the shape of the region to draw **graph1** over.

```
>
```

Exercise: What we have shown in this subsection is closely related to studying double integrals over non rectangular regions of real valued functions of two variables, something that is usually covered in the third semester of calculus. Look up this topic in a calculus book. Use Maple to duplicate as best you can some of the example pictures given in the book of functions defined over non rectangular regions.

```
>
```

```
>
```

5.6.3. Level curves, level sets, and the contourplot command

An important way to study a function of two variables is to look at its level curves. The **plot3d** command has the **style=patchcontour** option for drawing the graph of a function along with several of the function's level curves. Here is a graph of a bowl shaped surface along with several level curves.

```
> plot3d( 3*x^2+5*y^2, x=-5..5, y=-5..5, style=patchcontour,
  axes=framed );
>
```

If you rotate this graph so that the vertical axis is straight up and so that you are viewing the edge of the *xy*-plane, then you will see that the level curves are all parallel to the *xy*-plane. That is, each level curve is made of points on the surface that are all the same height above (or below) the *xy*-plane.

A **level curve** for the graph of a function $f(x, y)$ is the set of all points on the graph that have the same elevation.

Given a specific number c for the elevation, here is another way to define the level curve.

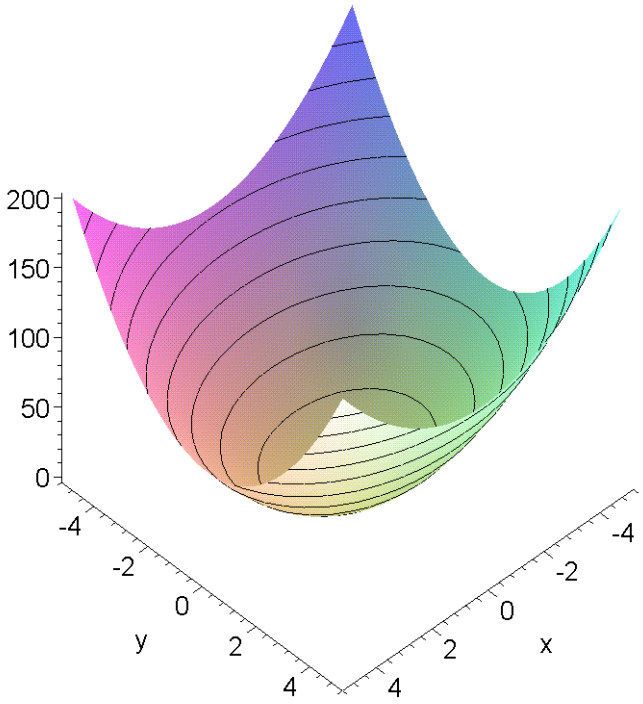
A **level curve with elevation** c *for* $f(x, y)$ *is the intersection of the graph of* $z = f(x, y)$ *with the horizontal plane* $z = c$.

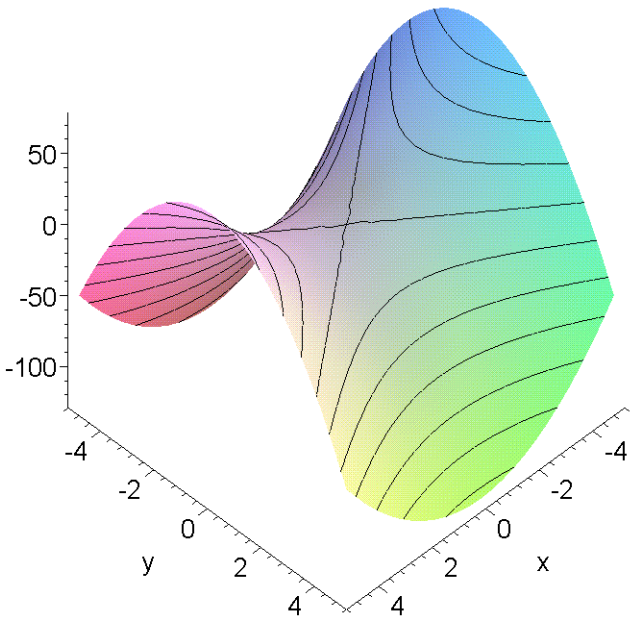
Notice that the level curves are part of the *graph* of a function. There is also the very closely related idea of a function's level sets, which are part of the function's *domain*.

A **level set** for a function $f(x, y)$ is the solution set of all points in the *xy*-plane that solve the equation $f(x, y) = c$ for some constant c.

[>

Level curves (and level sets) are an alternative way to provide visual information about the shape of a function's graph. For example, here are two graphs, one of a bowl shaped surface and one of a "saddle surface". If you rotate each of these two graphs, so that you are looking straight down the *z*-axis, then you will see two common and distinct patterns, one for the level curves around a local maximum (or minimum) and one for the level curves around a "saddle point".

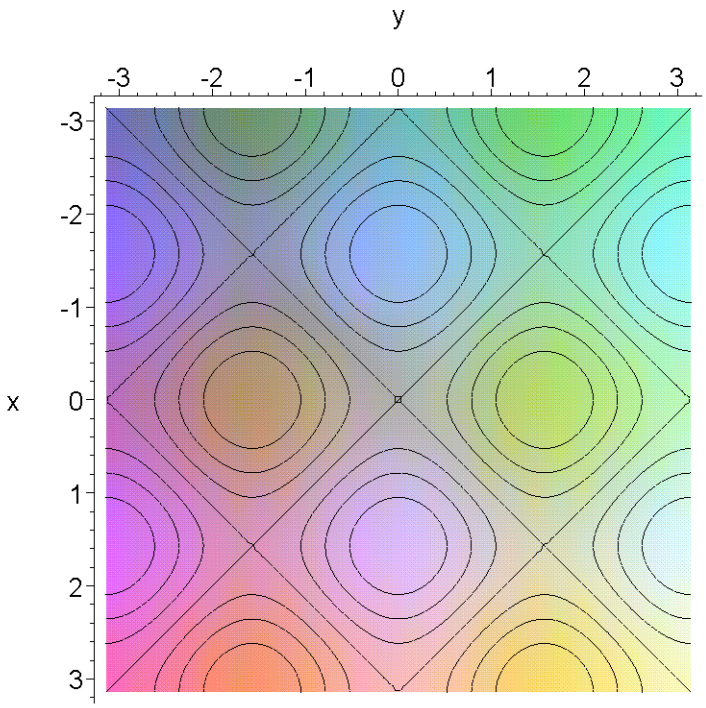




[**>**

Once you are used to thinking about and using level curves, you can look at a 2-dimensional graph of a function's level curves and "see" the shape of the graph by recognizing these distinct patterns. In the following graph we see an alternating pattern of saddle points and local maximums and minimums. Rotate this graph to see its 3-dimensional shape.

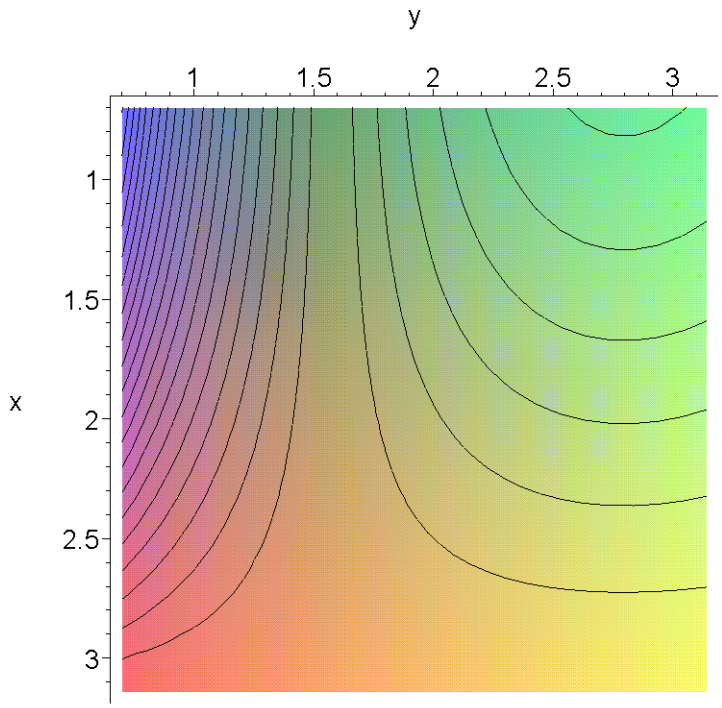
$$2\sin(x-y)\sin(x+y)$$



[>

Here is another common pattern that you will see with level curves. In the following graph, what do you think the closely spaced level curves mean, relative to the more spaced apart level curves? Rotate the graph to see its 3-dimensional shape.

$$\frac{|\sin(x)\cos(y)|}{x\,y}$$



Using a 2-dimensional picture to provide information about a 3-dimensional shape is especially useful in mapping, where topographic maps are the best way to describe complicated terrain to, for example, a hiker.

```
> 
```

Closely related to the idea of a function's level curves and level sets is the notion of a contour diagram for a function. A **contour diagram** for a function f(x, y) is made by drawing in the xy -plane a number of level sets for the function (or, what is nearly the same thing, by pushing the function's level curves down into the plane $z = 0$). So a contour diagram is very much like the two dimensional representations shown above for surfaces in three dimensions. Maple has the **contourplot** command from the **plots** package for drawing two dimensional contour diagrams for functions of two variables.

```
> x*y*exp(-(x^2+y^2));
```

```
> plots[contourplot]( %, x=-2..2, y=-2..2 );
```
Notice that the last graph is strictly two dimensional. You cannot rotate it.

We just mentioned that a contour diagram can be made by pushing a surface's level curves down into the plane $z = 0$. Here is a nice way to see this. The next command draws a three dimensional graph of the level curves for the same function as in the last command. But the next command sets the orientation of the graph so that we are looking straight down the z-axis onto the xy-plane (the plane where $z = 0$). So the next graph appears at first to be a two dimensional contour diagram (i.e., like the last graph). But you can rotate the next graph to see that the contours in this diagram are really floating in space. Looking straight down the z-axis has the visual affect of pushing the level curves down into the plane $z = 0$.

```
> plot3d( x*y*exp(-(x^2+y^2)), x=-2..2, y=-2..2, style=contour,
>            axes=normal, orientation=[-90,0] );
>
```

Here is another example of a contour plot and its 3-dimensional companion.

```
> -3*y/(x^2+y^2+1);
> plots[contourplot]( %, x=-10..10, y=-10..10, contours=17,
  axes=boxed );
>
> plot3d( -3*y/(x^2+y^2+1), x=-10..10, y=-10..10,
  style=contour, contours=17,
>            axes=boxed, orientation=[-90,0] );
>
```

The **contourplot** command has several options. For example, the following command specifies more contour lines, it specifies that the area between the contours should be shaded in, and the shading should shift from green to blue. (The default shading is from red, for low contour values, to yellow, for high contour values.) The shading helps to distinguish between valleys (local minimums) and peaks (local maximums) in the graph of the surface.

```
> plots[contourplot]( x*y*exp(-(x^2+y^2)), x=-2..2, y=-2..2,
>                     contours=15, filled=true,
  coloring=[green,blue] );
>
> plots[contourplot]( -3*y/(x^2+y^2+1), x=-10..10, y=-10..10,
>                      contours=35, filled=true,
  coloring=[blue,red], grid=[100,100] );
>
```

Drawing contour diagrams for real valued functions of two variables is closely related to graphing equations in two variables. To see the connection, consider a single level set, with say elevation c, for the function $f(x, y)$. Then the points from the xy-plane that are on that contour

are the points in the *xy*-plane that solve the equation $f(x, y) = c$. Here is an example. We will work with the following function.

```
> f := (x,y) -> x^2+y^2+sin(3*x)+sin(3*y);
```

Here is what its graph and some of its level curves look like.

```
> plot3d( f(x,y), x=-2..2, y=-2..2, style=patchcontour,
  axes=boxed );
```

Here is the same graph with a single level curve at elevation 2.

```
> plot3d( f(x,y), x=-2..2, y=-2..2, style=patchcontour,
  contours=[2], axes=boxed );
```

Here is the single level set (contour) with elevation 2.

```
> plots[contourplot]( f(x,y), x=-2..2, y=-2..2, contours=[2] );
```

Now look at the graph of the equation $f(x, y) = 2$.

```
> plots[implicitplot]( f(x,y)=2, x=-2..2, y=-2..2 );
```

The last two graphs are exactly the same.

We can use the above idea to mimic some of the functionality of the **contourplot** command and get a bit of an idea how this command might be implemented. The following command creates a sequence of equations of the form $f(x, y) = c_n$ where c_n takes on several different values.

```
> seq( f(x,y)=n, n=0..3 );
```

We can now use the **implicitplot** command to graph these four equations and get a contour diagram for **f**.

```
> plots[implicitplot]( {%}, x=-2..2, y=-2..2 );
```

Here is the same graph draw with **contourplot**.

```
> plots[contourplot]( f(x,y), x=-2..2, y=-2..2,
  contours=[0,1,2,3] );
```

So a contour diagram is exactly the same thing as drawing solutions to a sequence of "level set equations".

```
>
```

Exercise: Use the **seq** and **implicitplot** commands to duplicate the following contour diagram.

```
> plots[contourplot]( x*y*exp(-(x^2+y^2)), x=-2..2, y=-2..2 );
>
```

Maple has many different ways to draw the level curves and/or contours of a three dimensional surface. The **plot3d** command has a **contours** option for specifying the number of level curves to draw or for specifying the specific elevations to use for the level curves. Here is an example that requests three level curves.

```
> plot3d( 3*x^2+5*y^2, x=-10..10, y=-10..10,
  style=patchcontour, contours=3 );
```

Here is an example that specifies exactly which three level curves to draw.

```
> plot3d( 3*x^2+5*y^2, x=-5..5, y=-5..5, style=patchcontour,
  contours=[10,20,65] );
```

There is another option to **plot3d** that graphs *only* the surface's level curves. But from just the level curves it can be difficult to visualize the shape of a surface (unless you ask for a large number of level curves; try it).

```
> plot3d( 3*x^2+5*y^2, x=-5..5, y=-5..5, style=contour );
```

Here is an example of a more interesting surface and some of its level curves. Click on this graph with the mouse and look at the 3-D graphics context bar. There is a group of seven buttons near the middle of the context bar. These buttons let you switch between seven different **style** options. The third and fifth buttons (from the left) are for the styles **patchcontour** and **contour**. Try them.

```
> plot3d( 10*x*y*exp(-(x^2+y^2)), x=-2..2, y=-2..2,
  style=patchcontour );
```

Here are four Maple commands that draw four different visualizations of a function's level curves.

```
> f := (x,y) -> 5*x/(x^2 + y^2 + 1);
> plot3d( f(x,y), x=-3..3, y=-3..3, style=patchcontour,
  shading=z );
> plot3d( f(x,y), x=-3..3, y=-3..3, style=patchcontour,
  shading=z,
>                                    filled=true );
> plots[contourplot3d]( f(x,y), x=-3..3, y=-3..3,
  coloring=[blue,red],
>                                filled=true);
> plots[contourplot]( f(x,y), x=-3..3, y=-3..3,
  coloring=[blue,red],
>                               filled=true );
>
```

The next command uses a non rectangular domain so that the graph's level curves do not break up into pieces.

```
> plot3d( 3*x^2+5*y^2, x=-5..5,
  y=-sqrt(15-3/5*x^2)..sqrt(15-3/5*x^2 ),
>         style=patchcontour);
```

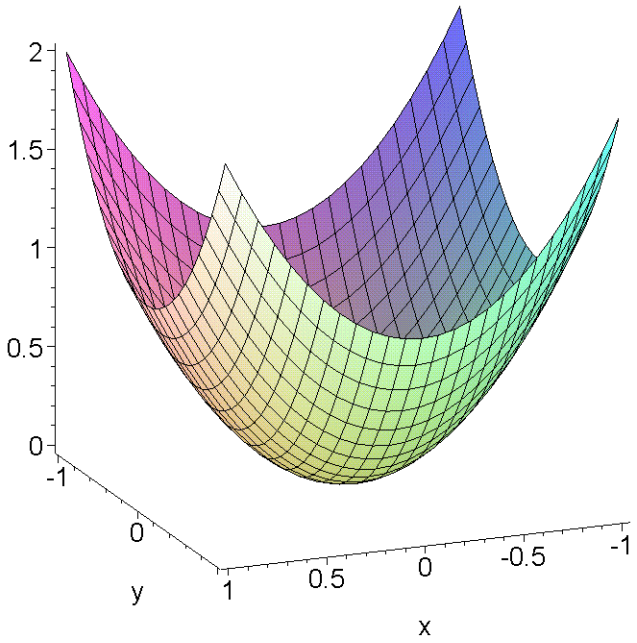
Notice that the idea of drawing a graph so that its top (or bottom) edge is "flat" is the same as drawing the graph so that its top (or bottom) edge is a level curve. We will say more about this in the next subsection.

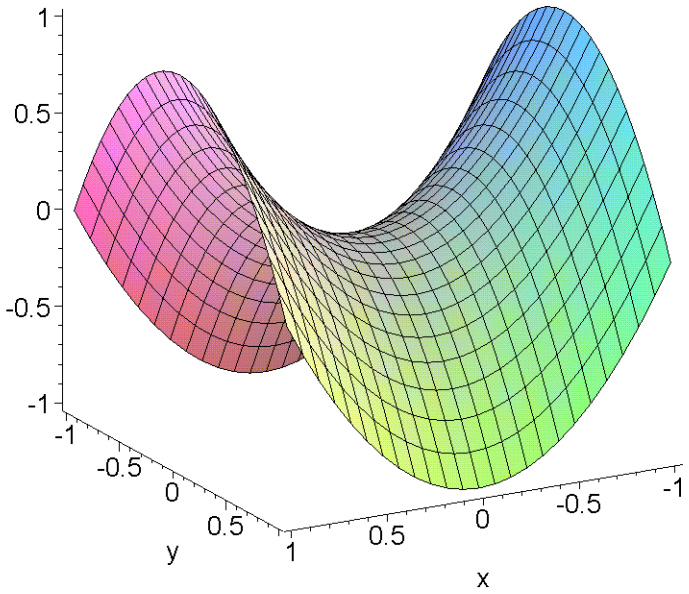
```
>
```

```
>
```

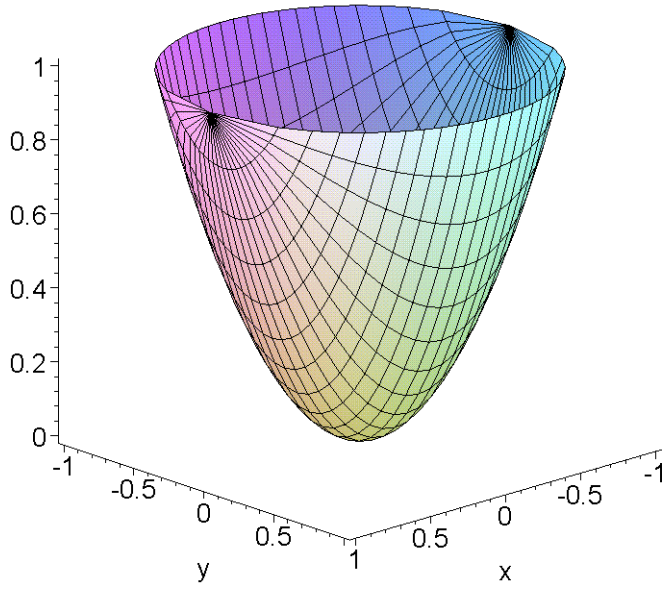
5.6.4. Using level curves when drawing surfaces

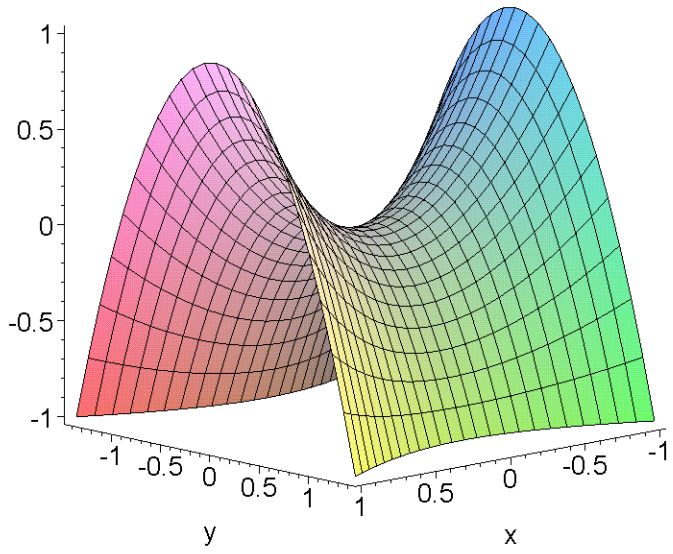
In a calculus textbook, you almost never see an elliptic paraboloid and a hyperbolic parabolioid graphed the following way.





Instead, you almost always see graphs that look something like these.





[**>**

The second two graphs are drawn so that the top and bottom edges (respectively) of the two graphs are level curves. You can verify this by clicking on the graphs and then using the context bar to change the style of each graph to **patchcontour**. If you click on all four graphs and rotate them so that you are looking straight down the *z*-axis, you can see what was changed to make the last two graphs look better. The first two graphs are drawn over rectangular regions and the second two graphs are drawn over non-rectangular regions. In this section we want to see how to choose non-rectangular regions so that the top and/or bottom edges of our graphs are level curves. This is a trick that is almost always used by textbooks to make their graphs of surfaces look better.

The first graph above was drawn by the following command.

[**> plot3d(x^2+y^2, x=-1..1, y=-1..1, axes=framed);**

Let us draw this graph so that its top edge is the level curve with elevation 1. To do this, we want the graph's non-rectangular region to be determined by the level set $x^2 + y^2 = 1$. We know that this is a circle of radius 1 centered at the origin. In order to make the interior of this level set into a non-rectangular region, we first need to decide if we shall use a Type I or a Type II region. We can use either, so let us use a Type I region. We need to find the two functions that describe the top and bottom edges of the Type I region. Let us have Maple solve the equation $x^2 + y^2 = 1$ for the two implicit functions that are these top and bottom edges.

[**> solve(x^2+y^2=1, y);**

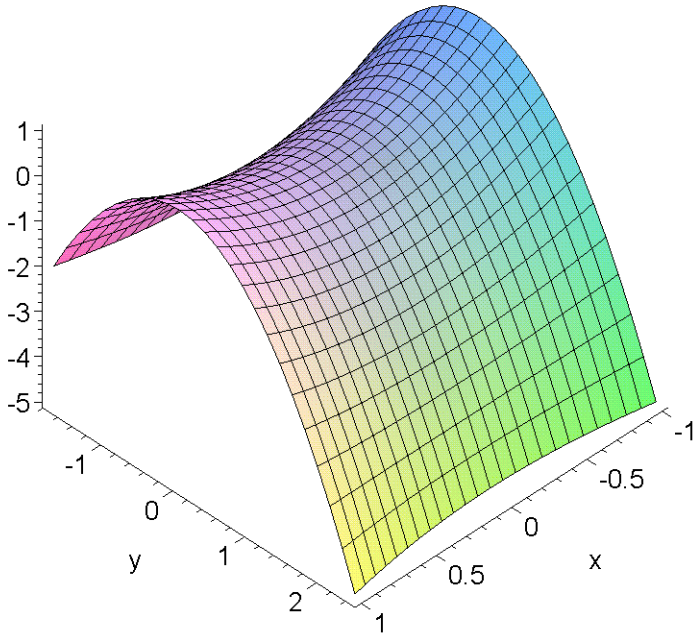
Let us name these two functions.

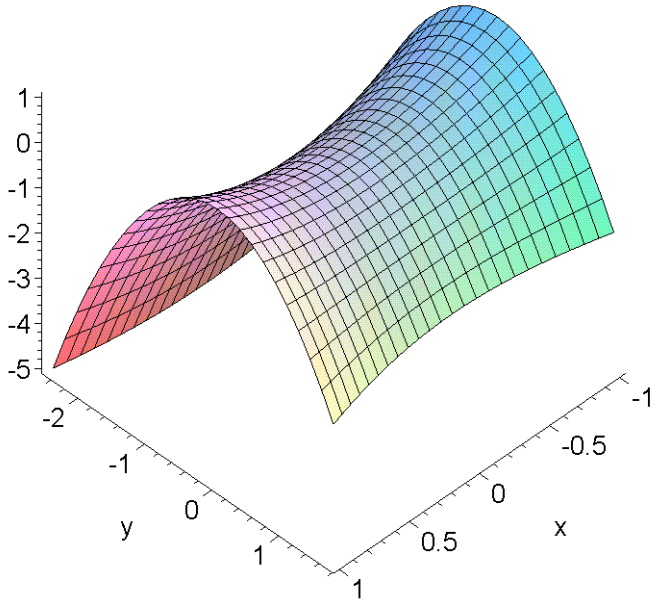
[

```
> y1 := %[1];
> y2 := %%[2];
```
Now draw the graph of our surface over the Type I region determined by these two functions.
```
> plot3d( x^2+y^2, x=-1..1, y=y1..y2, axes=framed );
>
```

Let's do the same for the hyperbolic paraboloid. The original graph was drawn by this command.
```
> plot3d( x^2-y^2, x=-1..1, y=-1..1, axes=framed );
```
By looking carefully at the numbers on the z-axis, we see that a good choice for the bottom edge level curve is $c = -1$. So we need to use the level set $x^2 - y^2 = -1$ to determine our non-rectangular region. If we imagine the horizontal plane $z = -1$ slicing the above graph, we can see that we will need to have our non-retangular region be a Type I region (we do not have a choice this time). We need to find the two functions that describe the top and bottom edges of this Type I region (these two functions are, essentially, where the horizontal plane $z = -1$ slices through the two sides of the saddle surface). Let us have Maple solve the equation $x^2 - y^2 = -1$ for the two implicit functions that are these top and bottom edges.
```
> solve( x^2-y^2=-1, y );
```
Let us name these two functions.
```
> y1 := %[1];
> y2 := %%[2];
```
Now draw the graph of our surface over the Type I region determined by these two functions.
```
> plot3d( x^2-y^2, x=-1..1, y=y1..y2, axes=framed );
>
```

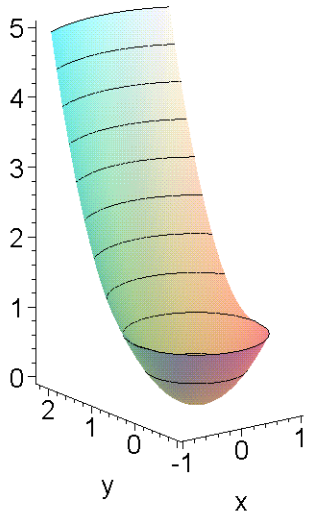
Exercise: Find the non-rectangular regions that reproduce the following two graphs.





[**>**

Exercise: Reproduce the following surface.




```
> 
```

Here is the graph of a plane over a rectangular region.

```
> plot3d( 3*x+5*y, x=-4..4, y=-2..6, axes=normal );
```

Let us redraw this plane with a bottom edge that is even with the *xy*-plane and a top edge that is parallel to the bottom edge. We need a Type I non-rectangular region that has the level set $3x + 5y = 0$ as the lower boundary of the region and the level set $3x + 5y = 20$ as the top boundary of the region. Let us have Maple compute these level sets for us.

```
> solve( 3*x+5*y= 0, y );
> solve( 3*x+5*y=20, y );
```

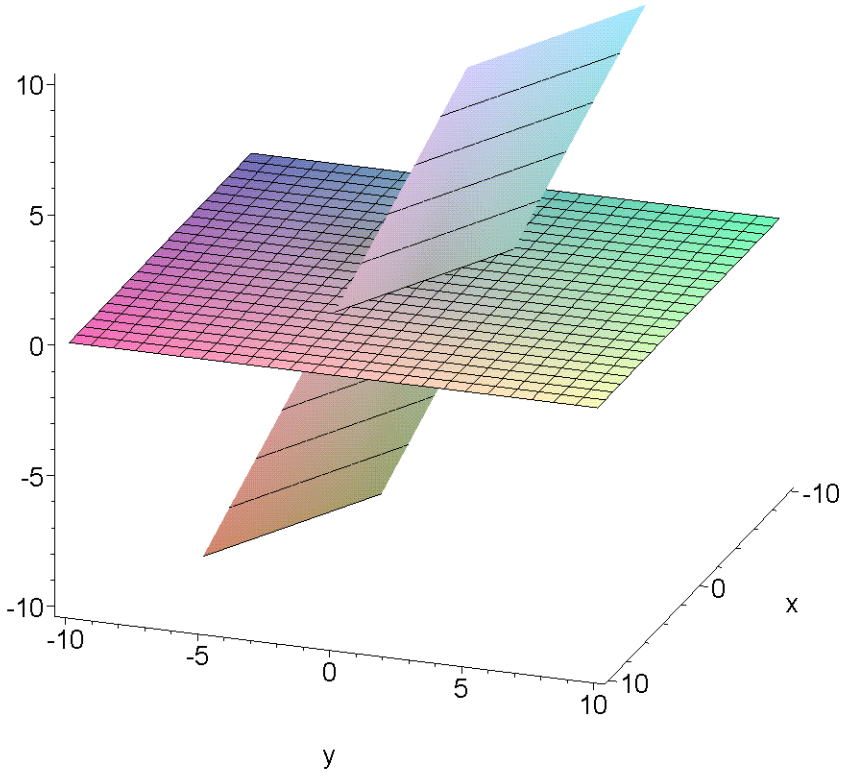
Here is the plane drawn over this region. Rotate both this graph and the above graph and see how the two graphs differ. In particular, rotate the graphs to look straight down the *z*-axis and see the shapes of the regions that the graphs are drawn over.

```
> plot3d( 3*x+5*y, x=-4..4, y=-3*x/5..-3*x/5+4, axes=normal );
```

For the sake of comparison, here is the first graph drawn with the two level curves that form the bottom and top edges of the second graph.

```
> plot3d( 3*x+5*y, x=-4..4, y=-2..6, style=patchcontour,
  contours=[0,20], axes=framed );
> 
```

Exercise: Create the following graph. The plane has equation $f(x, y) = x + 2y$.




```
> 
```

Let us work on a more complicated example. Consider the following graph.

```
> f := (x,y) -> -5*x/(1 + x^2 + y^2);
> plot3d( f(x,y), x=-5..5, y=-5..5, axes=framed );
```

Let us find a non-rectangular region that lets us isolate a region around the local maximum in the graph. In particular, let us find the non-rectangular region defined by the level curve with elevation 1, as shown in this graph (the elevation 1 level curve is the inner most of the three level curves).

```
> plot3d( f(x,y), x=-10..5, y=-5..5, style=patchcontour,
  contours=[1/2,3/4,1], axes=framed );
```

Let us have Maple solve for the two implicit functions defined by the level set $f(x, y) = 1$.

```
> g1 := solve( f(x,y)=1, y);
```

If we graph these two function, we get a sense of what our non-rectangular Type I region looks like.

```
> plot( [ g1[1], g1[2] ], x=-5..0 );
```

We need to know the left and right hand edges of our Type I region. So we need to solve for the zeros of our top and bottom boundary functions.

```
> a := solve( g1[1]=0, x );
```

Now we can graph our function over the non-rectangular Type I region.

```
> plot3d( f(x,y), x=a[1]..a[2], y=-g1[1]..g1[1], axes=framed );
```

There is a problem with the above graph. The upper and lower boundary functions of our Type I region have vertical tangent lines at the left and right edges of the region. Maple can have difficulties drawing graphs near vertical tangent lines. Here is a small trick that works in situations like this. We move one of our endpoints by a tiny, but to us imperceptible, amount.

```
> plot3d( f(x,y), x=a[1]..a[2]-0.001, y=-g1[1]..g1[1],
  axes=framed );
```

We have isolated the local maximum and "cut" it out of the rest of the surface. Let us "glue" this surface, with the isolated local maximum, back in to the original surface, but let's change its color at the same time. The next command redraws this local maximum in a different shading. The next two commands draw the original surface but without this local maximum (why do we need two commands?). The fourth command puts everything back together again.

```
> p1:=plot3d( f(x,y), x=a[1]..a[2]-0.001, y=-g1[1]..g1[1],
  shading=z ):
> p2:=plot3d( f(x,y), x=-6..6,
>                     y=piecewise(x<=a[1], 0, x<=a[2], g1[1],
  x>=a[2], 0)..5,
>                     grid=[100,100] ):
> p3:=plot3d( f(x,y), x=-6..6,
>                     y=-5..piecewise(x<=a[1], 0, x<=a[2],
  -g1[1], x>=a[2], 0),
>                     grid=[100,100] ):
> plots[display](p1,p2,p3, style=patchcontour, axes=framed);
>
```

Exercise: What do you see if you **display** plots **p2** and **p3** only? Why? How are plots **p2** and **p3** related to each other? Why do they need the **piecewise** expressions in them?

```
>
```

Here is the previous example again, but using a level set, $f(x, y) = \frac{1}{2}$, that encloses a larger region to isolate the local maximum.

```
> plot3d( f(x,y), x=-10..5, y=-5..5, style=patchcontour,
  contours=[1/2], axes=framed);
```

```
> g2 := solve( f(x,y)=1/2, y);
> b := solve( g2[1]=0, x );
> evalf(b);
> plot3d( f(x,y), x=b[1]..b[2], y=-g2[1]..g2[1],
  style=patchcontour, axes=framed);
>


>
```

5.6.5. Drawing curves on surfaces

In Sections 5.4 and 5.5 of this worksheet we worked with parametric curves. Let us see how we can combine parametric curves with graphs of functions in two variables. Let us draw some curves that lie on surfaces. The easiest way to draw a graph of a curve on a surface is to use the surface's function to "lift" a curve off of the plane and into the surface. Here is an example. We lift a spiral up to the graph of a paraboloid. This first plot draws the surface.

```
> g1 := plot3d( x^2+y^2, x=-2..2, y=-sqrt(4-x^2)..sqrt(4-x^2),


                style=hidden, shading=xy ):
```

The next plot draws the spiral lifted up into the surface.

```
> g2 := plots[spacecurve](
        [t*cos(20*t), t*sin(20*t),
  (t*cos(20*t))^2+(t*sin(20*t))^2],
        t=0..2, color=black, numpoints=200 ):
```

Now combine the two graphs.

```
> plots[display]( g1, g2 );
```

Try removing **g1** from the **display** command so that you see only the (lifted) spiral curve.

```
>
```

Here is the same example redone using Maple functions. Notice how in the definition of **g2** it is now more apparent what we mean by using the function **f** to lift the spiral curve up to the graph of the surface.

```
> f := (x,y) -> x^2+y^2;
> x := t -> t*cos(20*t);  # x coordinate of the spiral curve
> y := t -> t*sin(20*t);  # y coordinate of the spiral curve
> g1 := plot3d( f(x,y), x=-2..2, y=-sqrt(4-x^2)..sqrt(4-x^2),
                style=hidden, shading=xy ):
> g2 := plots[spacecurve](
            [ x(t), y(t), f(x(t),y(t)) ],
            t=0..2, color=black, numpoints=200 ):
> plots[display]( g1, g2 );
> unassign(f,x,y);
>
```

Exercise: The above example uses Maple functions but the two plot commands actually use the Maple functions to create expressions, so the plot comands are written using the syntax for expressions. Convert the two plot commands to use the syntax for Maple functions (in the **plot3d** command you will need to express the non rectangular domain using Maple functions and in the **spacecurve** command you will need to use the @ operator).

```
>
```

Here is the same example redone again using Maple functions in a slightly different way. In this example, the parameterization of the spiral curve is written as a vector valued function **h**. Notice how the notation used in the definition of **g2** has become much more compact.

```
> f := (x,y) -> x^2+y^2;
> h := t -> (t*cos(20*t), t*sin(20*t));  # the spiral curve
> g1 := plot3d( f(x,y), x=-2..2, y=-sqrt(4-x^2)..sqrt(4-x^2),
                style=hidden, shading=xy ):
> g2 := plots[spacecurve](
            [ h(t), f(h(t)) ],
            t=0..2, color=black, numpoints=200 ):
> plots[display]( g1, g2 );
> unassign(f,h,x,y);
>
```

Exercise: Redraw the last graph but with the spiral curve in the *xy*-plane added as a kind of "shadow" of its lifted version on the paraboloid. Draw the graph with both the surface in the graph and with the surface removed.

```
>
```

Exercise: Take one of the above examples of a curve drawn on a surface and convert the curve into a (pretty narrow) tube plot. Try this both with and without the surface itself in the graph.

```
>
```

Now let us look at an example from calculus of a curve on a surface. Recall that the partial derivative with respect to x of a function $f(x, y)$ at a point (x_0, y_0) is computed by holding y fixed at y_0 and then computing the ordinary derivative of $f(x, y_0)$ (which is a function of only one variable) at x_0. Geometrically, this means that we slice the graph of $f(x, y)$ through the point $(x_0, y_0, f(x_0, y_0))$ with a vertical plane parallel to the yz-plane, which gives us a curve in the slicing plane, and then compute the slope of the line tangent to this curve. Here is a picture of this for the function $f(x, y) = -x^2 - y^2$ and the point $(x_0, y_0) = (2, 2)$. Make sure you understand each step of this execution group and how it relates to the definition of the partial derivative.

```
> f := (x,y) -> -x^2-y^2;
> (x0, y0) := (2, 2);
> tl := x -> f(x0,y0)+D[1](f)(x0,y0)*(x-x0);
> g1 := plot3d( f, -5..5, -5..5 ):
```

```
> g2 := plots[implicitplot3d]( y = y0, x=-5..5, y=-5..5,
  z=-50..0 ):
> g3 := plots[spacecurve]( [t, y0, f(t,y0)], t=-5..5,
  color=black ):
> g4 := plots[spacecurve]( [t, y0, tl(t)], t=-2..5, color=black
  ):
> plots[display]( g1, g2, g3, g4, style=patchnogrid,
  axes=framed );
>
```

Exercise: Part (a) Modify the last example so that it demonstrates the partial derivative with respect to y at the point (x_0, y_0).

```
>
```

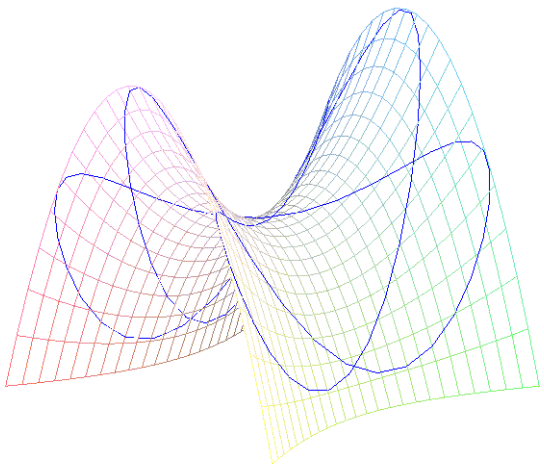
Part (b) Now combine the last two graphs and demonstrate both partial derivatives at the point (x_0, y_0).

```
>
```

Part (c) Add the tangent plane at the point (x_0, y_0) to your graph from part (b).

```
>
```

Exercise: Duplicate this graph. (Hint: The curve that is drawn on the surface is an example in Section 5.4.1.)



```
>
```

```
>
```

5.7. Graphs of parametric surfaces

Just as the **plot** command can graph both functions of one variable and also parametric curves in the plane, the **plot3d** command can graph both functions of two variables and parametric surfaces in three dimensional space. The following subsection goes over the syntax for using **plot3d** to graph parametric surfaces. The next two subsections go into quite a bit of detail about two common and important parameterizations, the sphere and the torus. The goal of these two subsections is to help you learn how to think about parametric surfaces and how to relate the properties of the three component functions with the geometric shape of the surface that they parameterize. The last subsection contains a number of exercises that make you think carefully about parametric surfaces. It also contains exercises that combine parametric curves in the plane with parametric surfaces to come up with unusual parameterizations of some interesting surfaces.

```
[ >
```

5.7.1. Parametric surfaces and the plot3d command

Before going over the details of parametric surfaces and the **plot3d** command, let us quickly review the case of the **plot** command.

Here is a graph of one function of a single variable.

```
[ > plot( cos(t), t=0..2*Pi );
```

Here is a graph of two functions, each of a single variable.

```
[ > plot( [ cos(t), sin(t) ], t=0..2*Pi );
```

Now if we move the range inside the brackets, the graph becomes a parametric curve in the plane (that is, the output only graph of a 2-dimensional vector valued function of a single variable), and the two functions are the horizontal and vertical component functions of the curve.

```
[ > plot( [ cos(t), sin(t), t=0..2*Pi ] );
[ >
```

Exercise: Here is a graph of three functions, each of a single variable.

```
[ > plot( [ cos(t), sin(t), 2*t ], t=0..2*Pi );
```

What if we now move the range inside the brackets?

```
[ > plot( [ cos(t), sin(t), 2*t, t=0..2*Pi ] );
```

How do we fix the last command so that it draws an appropriate curve?

```
[ >
```

Now let us look at the analogous use of the **plot3d** command. Here is a graph of a single function of two variables.

```
[ > plot3d( u*cos(v), u=-1..1, v=-Pi..Pi );
```

Here is a graph of two functions of two variables.

```
[ > plot3d( { u*cos(v), u*sin(v) }, u=-1..1, v=-Pi..Pi );
```

And here is a graph of three functions of two variables.

```
[ > plot3d( { u*cos(v), u*sin(v), v }, u=-1..1, v=-Pi..Pi );
```

Now replace the braces with brackets, and the graph becomes a parametric surface (that is, the output only graph of a 3-dimensional vector valued function of two variables), and the three functions become the *x*, *y*, and *z* components of the parameterization.

```
> plot3d( [ u*cos(v), u*sin(v), v ], u=-1..1, v=-Pi..Pi );
```

Notice a key difference in the syntax of the **plot** and **plot3d** commands. If we want to graph several functions of a single variable, the **plot** command allows us to place the list of functions inside either a pair of braces or a pair of brackets, but if we want to graph a parametric curve, we need the (two) functions along with their range inside of a pair of brackets. On the other hand, if we want to graph several functions of two variables, the **plot3d** command requires that we put the functions inside of a pair of braces, and if we want to graph a parametric surface we need the three functions inside of a pair of brackets and the two ranges outside of the brackets.

```
>
```

Exercise: Notice the error message that the following command produces.

```
> plot3d( [ u*cos(v), u*sin(v) ], u=-1..1, v=-Pi..Pi );
```

What do you think the error message means by "standard form" and "parametric form" for the "first argument"?

```
>
```

In the first section of this worksheet we said that a parametric surface is defined by a single function, a 3-dimensional vector valued function of two real variables. Here is a way to use a Maple function to emphasize that a parametric surface is really defined by a single (vector valued) function. The function $f(u, v) = (u\cos(v), u\sin(v), v)$ defines the surface in the previous example. Here is this function defined as a Maple function.

```
> f := (u,v) -> [ u*cos(v), u*sin(v), v ];
```

Here is how we use this function to graph the parametric surface.

```
> plot3d( f(u,v), u=-1..1, v=-Pi..Pi );
```

This example emphasizes that a parametric surface is defined by a single function. But it is usually more convenient to work with three expressions than with a single Maple function, so for the rest of this section we will express parametric surfaces by using three expressions for the three component functions of the parameterization.

```
>
```

Most third semester calculus books have several nice examples of interesting parametric surfaces. Here are a few examples that are typical of what you find in a calculus book.

```
> [ sin(u), u*sin(v), u*cos(v) ];
> plot3d( %, u=-Pi..Pi, v=0..3*Pi/2 );
```

```
> [ (u-sin(u))*cos(v), (1-cos(u))*sin(v), u ];
> plot3d( %, u=0..2*Pi, v=0..3*Pi/2 );
```

```
> [ (2+sin(v))*cos(u), (2+sin(v))*sin(u), u+cos(v) ];
```

```
> plot3d( %, u=0..4*Pi, v=0..2*Pi );

> [ (1-u)*(3+cos(v))*cos(4*Pi*u),
>   (1-u)*(3+cos(v))*sin(4*Pi*u),
>   3*u+(1-u)*sin(v) ];
> plot3d( %, u=0..1, v=0..2*Pi, orientation=[-14,76] );

> [ 2+cos(theta)+r*cos(theta/2),
>   2+sin(theta)+r*cos(theta/2),
>   r*sin(theta/2) ];
> plot3d( %, theta=0..2*Pi, r=-1/2..1/2, title="Mobius strip"
  );
>
```

Exercise: Redraw one of the last parametric surfaces using a single (vector valued) Maple function to define the parameterization.

```
>
```

Recall that in the previous section we mentioned that the **plot3d** command, when graphing a function of two variables, gives the x and y coordinates the preferred status of being the independent variables and **plot3d** always draws a graph of $z = \mathrm{f}(x, y)$. When graphing a parametric surface, **plot3d** does not give any coordinate direction a preferred status. It does however always treat the first expression after the opening bracket as the x-component, the second expression as the y-component, and the third expression as the z-component. We can use this to get other kinds of graphs from a function of two variables. That is, given a single real valued function f of two independent variables we can use parametric equations to draw any of the six graphs $z = \mathrm{f}(x, y)$, $z = \mathrm{f}(y, x)$, $y = \mathrm{f}(x, z)$, $y = \mathrm{f}(z, x)$, $x = \mathrm{f}(y, z)$, or $x = \mathrm{f}(z, y)$. For example, here is how we can graph $x = y^2 + z^2$ (as a parametric surface).

```
> plot3d( [y^2+z^2, y, z], y=-2..2, z=-2..2 );
```

Here is the analogous way to graph $z = x^2 + y^2$ as a parametric surface.

```
> plot3d( [x, y, x^2+y^2], x=-2..2, y=-2..2 );
>
```

Exercise: Use the scroll bar on the Maple window so that you can see both of the last two graphs at the same time. Use the mouse to rotate the two graphs into the same position. Then notice how similar mouse motions on each graph can cause different rotations of the graphs. Also, notice that each graph can be rotated around an axis that the other graph cannot be rotated about.

```
>
```

Here are all six possible graphs of the function $(u, v) \to u^2 + \sin(\pi v)$.

```
> f := (u,v) -> u^2+sin(Pi*v);
```

```
> plot3d( [x, y, f(x,y)], x=-2..2, y=-3..3 );
> plot3d( [x, y, f(y,x)], x=-2..2, y=-3..3 );
> plot3d( [x, f(x,z), z], x=-2..2, z=-3..3 );
> plot3d( [x, f(z,x), z], x=-2..2, z=-3..3 );
> plot3d( [f(y,z), y, z], y=-2..2, z=-3..3 );
> plot3d( [f(z,y), y, z], y=-2..2, z=-3..3 );
>
```

Exercise: Explain why the first, third, and fifth graphs above look similar. In what way do they differ from the second, fourth, and sixth graphs?

```
>
```

Exercise: Explain the relationships between the following four graphs. Notice that in each of the second, third, and fourth commands, a pair of **x**, **y** variables is switched relative to the first command.

```
> f := (u,v) -> u^2+sin(Pi*v);
> plot3d( [x, y, f(x,y)], x=-2..2, y=-3..3 );
> plot3d( [x, y, f(x,y)], y=-2..2, x=-3..3 );
> plot3d( [x, y, f(y,x)], x=-2..2, y=-3..3 );
> plot3d( [y, x, f(x,y)], x=-2..2, y=-3..3 );
>
```

Exercise: How would you graph two (or more) parametric surfaces at the same time, but with different parameter ranges for each surface? (Compare this with using **plot** or **spacecurve** to draw several parametric curves.)

```
>
```

```
>
```

5.7.2. Parameterizing a sphere

In the previous subsection we covered just the syntax of using the **plot3d** command to graph parametric surfaces. Let us try to understand something about how parametric surfaces work. A good starting place is to look at the standard parameterization of the sphere of radius one centered at the origin.

$$(\theta, \phi) \to [\sin(\phi)\cos(\theta), \sin(\phi)\sin(\theta), \cos(\phi)] \text{ with } \theta \in [0, 2\pi] \text{ and } \phi \in [0, \pi].$$

```
> [ sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi, title="Sphere" );
>
```

One way to understand this parameterization is to see the $\cos(\theta)$ and $\sin(\theta)$ terms as parameterizing a horizontal circle with a radius (which is the $\sin(\phi)$ term) that changes with the circle's height above (or below) the *xy*-plane. In other words, the parameterization defines the sphere as a stack of horizontal circles. As ϕ goes from 0 to π, the radius of these circles (i.e.,

sin(ϕ)) starts out at 0 when ϕ is 0 (at the "north pole"), the radius then grows to 1 when ϕ is $\pi/2$ (at the equator), and then shrinks back to 0 when ϕ reaches π (at the "south pole"). The height above the *xy*-plane of the circle that is being drawn is given by the cos(ϕ) term, which starts at 1 and decreases to -1 as ϕ goes from 0 to π.

Here is a way that helps us to visualize the roles played in the parameterization by the parameters θ and ϕ. The following parameterization changes the range for the variable ϕ from $0 .. \pi$ to $0 .. \frac{3\pi}{4}$. Since ϕ is supposed to determine the height of the horizontal circles and we restrict the range of ϕ, we should expect that some of the horizontal circles are missing. Rotate the surface drawn by the following commands and notice that the bottom of the sphere is missing.

```
> [ sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..3*Pi/4, title="Sphere" );
```

In the next parameterization, we restrict the range of θ to $0 .. \frac{11\pi}{6}$. Since θ is supposed to parameterize the horizontal circles and we restrict the range of θ, we should expect that every horizontal circle is missing some of its circumference, i.e, the sphere should have a vertical slice (or wedge) removed from it.

```
> [ sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..11*Pi/6, phi=0..3*Pi/4, title="Sphere" );
```

By further experimenting with the ranges for ϕ and θ, you should be able to build a sense of how these parameters are used to define the parametric surface.

```
>
```

Exercise: Draw a sphere with a band around the equator removed from the graph. (Hint: Use two parametric surfaces and the **display** command.)

```
>
```

Exercise: Draw a sphere with only four out of its eight octants so that no two of the drawn octants have an edge in common. (Hint: You need to use four **plot3d** commands and the **display** command.)

```
>
```

The standard parameterization of the sphere parameterizes the sphere as a stack of horizontal circles whose radii depended on their height above the *xy*-plane. If we modify the parameterization so that the radii are constant (say constantly equal to 1), then we get a stack of constant radii horizontal circles, i.e., a cylinder.

```
> [ 1*cos(theta), 1*sin(theta), cos(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi, title="Parametric
  Cylinder" );
```

Notice that with the parameterization the way it is now, the cylinder is drawn from the top

downward as the parameter ϕ varies from 0 to π and that the height of the cylinder is restricted between 1 and -1 (why?). If we change the third component of the parameterization from $\cos(\phi)$ to just ϕ, then the height of the cylinder can be defined arbitrarily.

```
> [ cos(theta), sin(theta), phi ];
> plot3d( %, theta=0..2*Pi, phi=-10..10, title="Parametric
  Cylinder" );
```

Use the **1:1** button on the graphics context bar to change the last two graphs and see that the second graph really is taller thatn the first.

```
>
```

Exercise: Convert the last parameterization into a parameterization of a cone. So the radii of the stacked circles will vary linearly with the height of the circles.

```
>
```

In the section on parametric curves, we saw that animations of the parameter sweeping out a curve were helpful in visualizing how the parameterization works. We can do something similar in the case of parametric surfaces. An animation of a parametric surface can "unfold" one parameter at a time. Such an animation can help us to understand the role played in the parameterization by that parameter.

But before doing that, let us review how we animate parametric curves. Recall from the section on animating parametric curves that the easiest way to "unfold" a parameterization of a curve is to use **animatecurve**. For example, the standard parameterization of a circle, $\theta \rightarrow [\cos(\theta), \sin(\theta)]$, can be animated like this.

```
> plots[animatecurve]( [cos(theta), sin(theta), theta=0..2*Pi]
  );
```

We can also do this "unfolding" of the parameterization by using a trick with the **animate** command.

```
> plots[animate]( [cos(s*theta), sin(s*theta), theta=0..2*Pi],
  s=0..1 );
```

The animation parameter **s** rescales the parameter **theta** in the component functions of the parameterization.

```
>
```

Maple does not have an "animatesurface" command that would be analogous to **animatecurve**. But Maple does have an **animate3d** command that is analogous to **animate**. To unfold the parameterization of a surface we use the **animate3d** command in a manner similar to the above use of **animate**.

Let us return to the standard parameterization of the sphere,

$$(\theta, \phi) \rightarrow [\sin(\phi)\cos(\theta), \sin(\phi)\sin(\theta), \cos(\phi)]$$

```
> [ sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
```

```
> plot3d( %, theta=0..2*Pi, phi=0..Pi, title="Sphere" );
```

and let us animate this parameterization by unfolding the parameterization in each of the θ and ϕ "directions".

The next animation unfolds the parameterization in the θ direction and shows the rotation caused by the θ parameter as it sweeps out horizontal circles. Here is the animation created using **animate3d**.

```
> (theta,phi) -> [sin(phi)*cos(theta), sin(phi)*sin(theta),
  cos(phi)];
> plots[animate3d]( %(s*theta, phi), theta=0..2*Pi, phi=0..Pi,
>                   s=0..1, frames=60, orientation=[-60,60],
>                   title="Animated Sphere");
>
```

The next animation unfolds the parameterization in the ϕ direction and shows how the ϕ parameter determines both the height and the radius of the circles swept out by the θ parameter.

```
> (theta,phi) -> [sin(phi)*cos(theta), sin(phi)*sin(theta),
  cos(phi)];
> plots[animate3d]( %(theta, s*phi), theta=0..2*Pi, phi=0..Pi,
>                   s=0..1, frames=60, orientation=[30,100],
>                   scaling=constrained, title="Animated
  Sphere");
>
```

The next animation shows how we can unfold both parameters at once. The resulting animation is interesting to watch, but it probably does not help one to understand the parameterization as much as the previous two animations.

```
> (theta,phi) -> [sin(phi)*cos(theta), sin(phi)*sin(theta),
  cos(phi)];
> plots[animate3d]( %(s*theta, s*phi), theta=0..2*Pi,
  phi=0..Pi,
>                   s=0..1, frames=60, orientation=[-60,90],
>                   scaling=constrained, title="Animated
  Sphere");
>
```

Exercise: Here are two parameterizations of a sphere. The standard parameterization.

```
> [ sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi );
```

And an alternative parameterization.

```
> [ cos(phi)*sin(theta), cos(phi)*cos(theta), sin(phi) ];
> plot3d( %, theta=0..Pi, phi=0..2*Pi );
```

The standard parameterization can be described as creating the sphere as a "half circle's worth of horizontal circles" while the alternative parameterization can be described as creating the sphere as a "circle's worth of horizontal half circles".
Part (a): Use animations to discover how these parameterizations differ in the way that they unfold each parametric direction. Explain the idea behind each parameterization's description.

```
>
```

Part (b): Change the standard parameterization into a parameterization that creates a "circle's worth of horizontal half circles". Change the alternative parameterization into a parameterization that creates a "half circle's worth of horizontal circles". (Hint: For both of these changes, you only need to change the ranges used in each parameterization.)

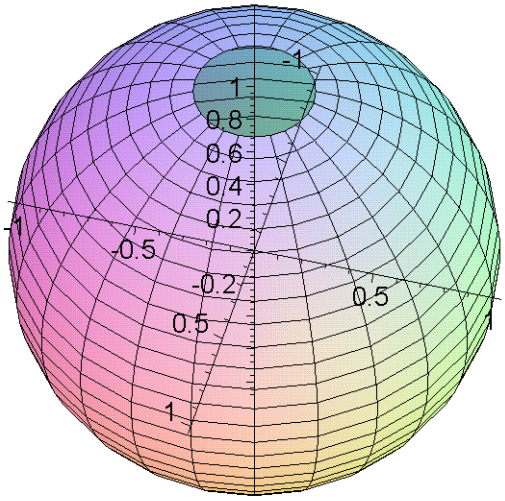
```
>
```

Exercise: In the standard parameterization of the sphere, the third component of the parameterization, $\cos(\phi)$, determines the height of a horizontal circle. As ϕ ranges from 0 to π, the height goes from 1 to -1. In the following parameterization, the height function has been replaced with the linear function $1 - \frac{2\phi}{\pi}$, which also goes from 1 to -1 as ϕ ranges from 0 to π. Explain why the parametric surface has the shape that it does.

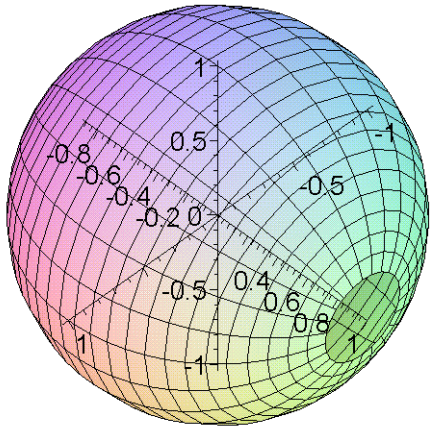
```
> [ sin(phi)*cos(theta), sin(phi)*sin(theta), 1-2*phi/Pi ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi, title="Christmas
  Ornament" );
```

```
>
```

Exercise: Use the standard parameterization of a sphere to draw the following surface. The hole at the top of the sphere has radius 1/4 and the hole at the bottom of the sphere has radius 1/2.



Exercise: Find a parameterization of the sphere that will let you draw the following surface. Note that the holes in the suface must be centered on the y-axis. The radius of the hole in front is 1/4 and the radius of the hole in the back is 1/2.



Exercise: Explain why the following parameterization produces the shape that it does.

```
> [ (1-abs(phi))*cos(theta), (1-abs(phi))*sin(theta), phi ];
> plot3d( %, theta=0..2*Pi, phi=-1..1 );
>
```

Exercise: Explain why the following parameterization produces the shape that it does.

```
> [ cos(phi)*cos(theta), cos(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi,
>         title="Modified sphere parameterization" );
>
```

Exercise: The following surface has a circular profile when you look straight down one coordinate axis, it has a cone shaped profile when you look straight down another coordinate axis, and it has a square profile when you look straight down the third coordinate axis. Use the parameterization to explain these profiles.

```
> [ cos(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi,
>         title="Modified sphere parameterization" );
>
```

```
>
```

5.7.3. Parameterizing a torus

Let us look at another important example of a parametric surface and try to understand how the parameterization works. The following is a parameterization of a torus (that is, a donut, or inner tube, shape).

$$(\theta, \phi) \to ((2 + \cos(\phi)) \cos(\theta), (2 + \cos(\phi)) \sin(\theta), \sin(\phi))$$

```
> [ (2+cos(phi))*cos(theta),
>   (2+cos(phi))*sin(theta),
>   sin(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..2*Pi,
>         scaling=constrained, title="Torus" );
>
```

There are several ways to understand this parameterization. First, let us look at it in a way similar to how we analyzed the parameterization of the sphere. The first two components parameterize horizontal circles with radius given by the expression $2 + \cos(\phi)$. Notice that the maximum radius is 3 and the minimum radius is 1. The third component determines the height of a horizontal circle. Notice that the height will start at 0 (when $\phi = 0$) and then the height will rise up to 1, drop back down to 0, keep dropping down to -1, and then rise back up to 0. So any particular height will be attained at two different values for the parameter ϕ. But the two different values of ϕ that determine the same height will determine different radii for the horizontal circles. If we think of ϕ as determining horizontal circles of different heights and different radii, the image that we should get is that as the height starts out at 0 and rises, the radius of the horizontal circle starts with value 3 and decreases as the circle rises. When the height gets to its maximum of 1, the radius is 2 and it continues to decrease. As the height drops back down towards 0, the horizontal circles are smaller that the circles drawn when the height was increasing to 1. When the height gets back to zero, the radius is 1 and the top half of the torus is complete. As the height continues to decrease, the radius begins to increase. When the height is at its minimum of -1, the radius is 2. As the height begins to increase again, the radius continues to increase. When the height reach 0 again, the radius is back to 3 and the bottom of the torus is complete.

Let us confirm this analysis by making some changes in the ranges of the parameters as we did with the sphere parameterization. According to the above analysis, if we restrict the range of ϕ to $0 .. \pi$, we should get just the upper half of the torus.

```
> [ (2+cos(phi))*cos(theta), (2+cos(phi))*sin(theta), sin(phi)
  ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi,
>         title="Torus", scaling=constrained );
```

If we restrict the range of ϕ to be $\frac{\pi}{2} .. \frac{3\pi}{2}$, then we should get the "inner" half of the torus.

```
> [ (2+cos(phi))*cos(theta), (2+cos(phi))*sin(theta), sin(phi)
```

```
   ];
> plot3d( %, theta=0..2*Pi, phi=Pi/2..3*Pi/2,
>         title="Torus", scaling=constrained );
```

Since θ is supposed to be parameterizing the horizontal circles, restricting the range of θ should remove a sector from the torus. If we restrict the range of θ to be $\frac{\pi}{4}$ .. 2 π, we should remove the first eight of the circumference of the torus.

```
> [ (2+cos(phi))*cos(theta), (2+cos(phi))*sin(theta), sin(phi)
   ];
> plot3d( %, theta=Pi/4..2*Pi, phi=0..2*Pi,
>         title="Torus", scaling=constrained );
>
```

Exercise: Draw the torus with a band around it two "equators" removed, so the torus will be cut into an upper and a lower half.

```
>
```

Exercise: Draw the torus with only four out of its eight octants so that no two of the drawn octants have an edge in common. (Hint: You need to use four **plot3d** commands and the **display** command.)

```
>
```

Here is another way to understand the parameterization of the torus. First of all, a torus is a "circle's worth of circles". We can define a torus by starting with a circle (of radius 2) lying in the *xy*-plane and centered at the origin, and then for each point *p* on this circle, we draw a vertical circle with its center at *p* and lying in the vertical plane determined by *p* and the origin. In other words, a circle's worth of circles. We can use this way of describing a torus to give a different analysis of our parameterization of the torus. The "circle of centers" in the *xy*-plane is parameterized by

$$p(\theta) = [2\cos(\theta), 2\sin(\theta), 0].$$

For each choice of θ we get a different point p(θ) on this circle. The curve [cos(ϕ), 0, sin(ϕ)] parameterizes a vertical circle centered at the origin in the *xz*-plane. If we apply a rotation matrix to the points on this vertical circle

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\phi) \\ 0 \\ \sin(\phi) \end{bmatrix}$$

we get

$$[\cos(\theta)\cos(\phi), \sin(\theta)\cos(\phi), \sin(\phi)]$$

which parameterizes a vertical circle centered at the origin in the vertical plane determined by the direction of p(θ) (notice that we have also re-derived the parameterization of a sphere; why?). If we take this parameterization of a vertical circle and add it to the parameterization of the circle of centers, p(θ),

$$[2\cos(\theta), 2\sin(\theta), 0] + [\cos(\theta)\cos(\phi), \sin(\theta)\cos(\phi), \sin(\phi)]$$

then we take each vertical circle and move its center to a point on the circle of centers. But the last sum is another way of writing our parameterization of the torus.

```
> [2*cos(theta), 2*sin(theta), 0]
>   + [cos(theta)*cos(phi), sin(theta)*cos(phi), sin(phi)];
> factor(%);
```

In summary, choosing a value for θ in the parameterization chooses a point on the circle of centers and then the parameter ϕ traces out one of the vertical circles in the "circle's worth of circles".

```
>
```

The following graph illustrates this way of thinking about a torus. The graph draws the "circle of centers" in the *xy*-plane, then it draws three vertical circles centered at three different points *p* around the "circle of centers", and then the graph draws part of the torus as it goes around the rest of the "circle of centers".

```
> g1 := plots[spacecurve]([ 2*cos(theta), 2*sin(theta), 0 ],
>                          theta=0..2*Pi, color=black):
> g2 := plots[spacecurve]([ 2*cos(0)+cos(0)*cos(phi),
>                          2*sin(0)+sin(0)*cos(phi),
>                          sin(phi) ], phi=0..2*Pi,
  color=black ):
> g3 := plots[spacecurve]([ 2*cos(Pi/4)+cos(Pi/4)*cos(phi),
>                          2*sin(Pi/4)+sin(Pi/4)*cos(phi),
>                          sin(phi) ], phi=0..2*Pi, color=blue
  ):
> g4 := plots[spacecurve]([ 2*cos(-Pi/4)+cos(-Pi/4)*cos(phi),
>                          2*sin(-Pi/4)+sin(-Pi/4)*cos(phi),
>                          sin(phi) ], phi=0..2*Pi, color=red
  ):
> torus := [(2+cos(phi))*cos(theta), (2+cos(phi))*sin(theta),
  sin(phi)]:
> g5 := plot3d( torus, theta=Pi/2..3*Pi/2, phi=0..2*Pi ):
> plots[display]( g1, g2, g3, g4, g5, scaling=constrained,
>                 axes=normal, style=patchnogrid );
>
```

Here is still another way to derive the parameterization of the torus. This third way is similar to the second way. Start by noticing that the torus is a surface of revolution. We get a torus by starting with a unit circle in the *xz*-plane with center (2,0,0) and then revolving this circle around the *z*-axis. The unit circle in the *xz*-plane with center at (2,0,0) has the following parameterization.

$$[2 + \cos(\phi), 0, \sin(\phi)]$$

If we apply a rotation matrix to the points on this vertical circle

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2+\cos(\phi) \\ 0 \\ \sin(\phi) \end{bmatrix}$$

then we get our parameterization of the torus. Let us do this matrix multiplication using Maple. Here is the rotation matrix.

```
> Matrix( [ [cos(theta), -sin(theta), 0],
>           [sin(theta),  cos(theta), 0],
>           [         0,           0, 1] ] );
```

Here is the vector that represents the points on the unit circle in the *xz*-plane with center at (2,0,0).

```
> Vector( [2+cos(phi), 0, sin(phi)] );
```

Here is how we compute their product.

```
> %% . %;
```

The next command rewrites the last result in the form that we usually use for surface parameterization, as a "row vector".

```
> LinearAlgebra[Transpose](%);
>
```

Let us create animations that unfold the standard parameterization of the torus.

$$(\theta, \phi) \rightarrow ((2+\cos(\phi))\cos(\theta), (2+\cos(\phi))\sin(\theta), \sin(\phi))$$

The next animation can be thought of in two ways. If we think of the parameterization of the torus as a stack of horizontal circles, then this animation shows how the circles are stacked as a function of their height. The animation starts with a horizontal circle at height 0 (since $\sin(\phi) = 0$ when $\phi = 0$), then it draws the circles with increasing height until $\phi = \frac{\pi}{2}$, then it draws circles with decreasing height until $\phi = \frac{3\pi}{2}$, and then draws circles with height increasing back to 0 when $\phi = 2\pi$. On the other hand, if we think of the parameterization as a circle's worth of circles, then this animation animates the parameterization of all the vertical circles simultaneously.

```
> (theta,phi) -> [(2+cos(phi))*cos(theta),
>                 (2+cos(phi))*sin(theta),
>                 sin(phi)];
> plots[animate3d]( %(theta, s*phi), theta=0..2*Pi,
  phi=0..2*Pi,
>                   s=0..1, frames=60, orientation=[90,130],
>                   scaling=constrained, title="Animated
  Torus");
>
```

Here is another view of this parameterization. In this version, only half of the torus is drawn to

make it easier to see how the animation is unfolding the parameterization in the ϕ direction around the vertical circles.

```
> (theta,phi) -> [(2+cos(phi))*cos(theta),
>                 (2+cos(phi))*sin(theta),
>                 sin(phi)];
> plots[animate3d]( %(theta, s*phi), theta=0..Pi, phi=0..2*Pi,
>                   s=0..1, frames=60, orientation=[-120,70],
>                   scaling=constrained, title="Animated Half
  Torus");
>
```

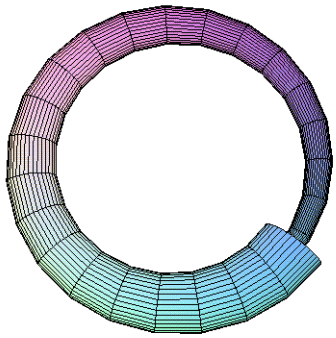
The next toral animation animates going around the circle of centers. That is, this animation unfolds the parameterization in the θ direction around the circle of centers.

```
> (theta,phi) -> [(2+cos(phi))*cos(theta),
>                 (2+cos(phi))*sin(theta),
>                 sin(phi)];
> plots[animate3d]( %(s*theta, phi), theta=0..2*Pi,
  phi=0..2*Pi,
>                   s=0..1, frames=60, orientation=[-90,60],
>                   scaling=constrained, title="Animated
  Torus");
>
```

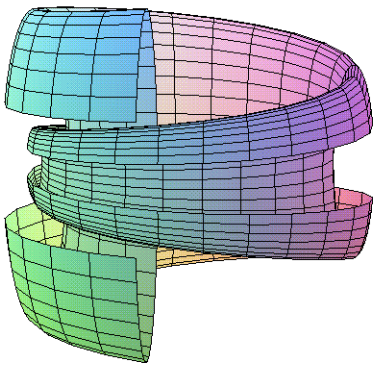
The third toral animation combines the previous two and it unfolds both parametric directions at the same time.

```
> (theta,phi) -> [(2+cos(phi))*cos(theta),
>                 (2+cos(phi))*sin(theta),
>                 sin(phi)];
> plots[animate3d]( %(s*theta, s*phi), theta=0..2*Pi,
  phi=0..2*Pi,
>                   s=0..1, frames=60, orientation=[-110,-160],
>                   scaling=constrained, title="Animated
  Torus");
>
```

Exercise: Modify the parameterization of the torus to create a parameterization of the following surface.



Here is another illustration of this surface, with part of the wall of the surface cut away so that you can see how the surface wraps inside of itself.



Hint: Go back to the analysis of the torus as a "circle's worth of vertical circles". In this way of looking at the parameterization, you want to make the radii of the vertical circles change with their location around the horizontal circle.

[**>**

[>

5.7.4. Parameterizations as patterns

In this section we consider the parameterization of the sphere and the torus as patterns that we can use to generate new parameterizations for a variety of surfaces. For example, from the standard parameterization for the sphere we can genertate parameterizations for a cube, pyramid, tetrahedron, cone, and prism. We do this by analyzing the role played by each function in the parameterization of the sphere and then using that information to tell us how we can replace those functions with component functions from parameteric curves.

[>

Let us look once again at the parameterization of a sphere. Horizontal cross sections of a sphere are circles. So we can think of a sphere as a stack of circles whose radii vary with their height above the *xy*-plane. Now look again at the component functions of the parameterization.

$$x = \sin(\phi)\cos(\theta), \qquad y = \sin(\phi)\sin(\theta), \qquad z = \cos(\phi).$$

Notice that for a fixed value of ϕ, the *x* and *y* coordinates parameterize a horizontal circle of radius $\sin(\phi)$ that has height $\cos(\phi)$ above the *xy*-plane. So we see right away that this parameterization does describe the sphere as a stack of horizontal circles. The height above the *xy*-plane of a circle is given by the $\cos(\phi)$ term which starts at 1 and decreases to -1 as ϕ goes from 0 to π. The radius term $\sin(\phi)$ starts out at 0 when ϕ is 0 (at the "north pole") and the radius then grows to 1 and then shrinks back to 0 as ϕ goes from 0 to π. The way the radii of these circles change with their height is described by the profile of the sphere when the sphere is cut by a vertical plane. If we fix $\theta = 0$, then $y = 0$ and we see that the profile of the sphere in the vertical *xz*-plane is parameterized by $x = \sin(\phi)$ and $z = \cos(\phi)$ with $\phi \in [0, \pi]$, which is clearly a half circle profile (similarly for fixed $\theta = \frac{\pi}{2}$ and the *yz*-plane).

[>

In summary, we can say that the $\cos(\theta)$ and $\sin(\theta)$ terms in the paramterization of the sphere parameterize a curve (i.e., a circle) which is the horizontal cross section of the surface, and the $\sin(\phi)$ and $\cos(\phi)$ terms parameterize a curve (i.e., a half circle) which is the vertical profile of the surface. Now let us see how we can use this interpretation of the parameterization of the sphere to create parameterizations of some new surfaces.

In Section 5.4.4 we created parameteriztions for a few simple polygons like the square and the triangle. We showed in that section how the components of those parameteriztions are very much like the cosine and sine functions in the parameterization of the circle. What we shall do is take the component functions from the polygon parameterizations and use them to replace appropriate cosine and sine functions in the sphere parameterization, thereby changing the curves that are the horizontal and/or vertical cross sections of the parameterization. By choosing different horizontal and vertical cross sections, we can modify the parameterization of the sphere into parameterizations of many different surfaces.

[

```
>
```

For our first example, let us change the vertical profile in the sphere parameterization from a half circle into a half triangle. Here are horizontal and vertical component functions for the parameterization of an equilateral triangle.

```
> tri_horz := t -> piecewise( t < 1/3, -4.5*t+1,
>                             t < 2/3, -0.5,
>                             t <= 1,   4.5*t-3.5 );
> tri_vert := t -> piecewise( t < 1/3,  2.598*t,
>                             t < 2/3, -5.196*t+2.598,
>                             t <= 1,   2.598*t-2.598 );
```

Here is the curve that these functions parameterize.

```
> plot( [tri_horz(t), tri_vert(t), t=0..1] );
```

These component functions parameterize the triangle in a manner similar to the way that the cosine and sine functions parameterize a circle (with `tri_vert` being analogous to cosine and `tri_vert` being analogous to sine). In particular, the parameterization starts when $t = 0$ at the point (1,0), the parameterization goes around the triangle counter clockwise, and the range of t from 0 to 1/2 parameterizes the upper half of the triangle.

```
> plots[animatecurve]( [tri_horz(t), tri_vert(t), t=0..1/2],
>                      view=[-1..1,-1..1] );
```

Let us "plug in" these two component functions in place of the $\cos(\phi)$ and $\sin(\phi)$ terms from the paramterization of the sphere. Recall that these two terms gave the sphere it half circle profile. The new parameterized surface will have circular horizontal cross sections and a triangular profile. (Notice that the range for ϕ is changed to 0 to 1/2, instead of 0 to π).

```
> '[tri_vert(phi)*cos(theta), tri_vert(phi)*sin(theta),
  tri_horz(phi)]';
> plot3d( %, theta=0..2*Pi, phi=0..1/2 );
>
```

Exercise: How would you remove the bottom from the previous surface to create an open cone?

```
>
```

Exercise: Consider the following two piecewise defined functions.

```
> f := x -> piecewise( x<1/4, 4*x, x<3/4, 1, x<=1, -4*x+4 );
> g := x -> piecewise( x<1/4, 1, x<3/4, 2-4*x, x<=1, -1 );
```

Notice that they parameterize three out of four sides of a square in the plane.

```
> plot( [ f(t), g(t), t=0..1] );
```

The following parameterization draws a "can". Explain how this parameterization works by comparing it to the standard parameterization of a sphere. Also, how would you make this can's height twice what its diameter is? How would you turn this into a parameterization of a can with no lid?

```
> '[ f(phi)*cos(theta), f(phi)*sin(theta), g(phi) ]';
```

```
> plot3d( %, theta=0..2*Pi, phi=0..1, title="Can" );
>
```

Exercise: Here is a parameterization of a closed frustum of a cone that has radius 1 at the top and radius 2 at the bottom and height 1 (the height is the vertical distance from the top surface to the bottom surface). This parameterization is based on the previous exercise and its parameterization of a can. Modify this parameterization so that it uses three parameters, **a**, **b**, and **c**, such that **a** and **b** determine the radius of the top and bottom of the frustum respectively and **c** is the height of the frustrum.

```
> f := x -> piecewise( x<1/4, 4*x, x<3/4, 2*x+1/2, x<=1, -8*x+8
  );
> g := x -> piecewise( x<1/4, 1, x<3/4, 3/2-2*x, x<=1, 0 );
> '[ f(phi)*cos(theta), f(phi)*sin(theta), g(phi) ]';
> plot3d( %, theta=0..2*Pi, phi=0..1, title="Frustum of a Cone"
  );
>
```

Now let us take our cone parameteriztion and replace the circular cross sections with square cross sections. We need to replace the cos(θ) and sin(θ) terms with the components of a parameterization of a square.

```
> sq_horz := t -> piecewise( t <= 1/8,  1,
>                            t <= 3/8, -8*t+2,
>                            t <= 5/8, -1,
>                            t <= 7/8,  8*t-6,
>                            t <= 1,    1 );
> sq_vert := t -> piecewise( t <= 1/8,  8*t,
>                            t <= 3/8,  1,
>                            t <= 5/8, -8*t+4,
>                            t <= 7/8, -1,
>                            t <= 1,    8*t-8 );
```

Here is a parameterization of the surface with square horizontal cross sections and a triangular profile. (Notice that the range for θ is changed to 0 to 1 from 0 to 2 π.)

```
> '[ tri_vert(phi)*sq_horz(theta),
>    tri_vert(phi)*sq_vert(theta),
>    tri_horz(phi) ]';
> plot3d( %, theta=0..1, phi=0..1/2 );
>
```

Here is the surface with a square profile and circular cross sections (again).

```
> '[sq_vert(phi)*cos(theta), sq_vert(phi)*sin(theta),
  sq_horz(phi)]';
> plot3d( %, theta=0..2*Pi, phi=0..1/2 );
```

```
>
```

The surface with a square profile and square cross sections.

```
> '[ sq_vert(phi)*sq_horz(theta),
>    sq_vert(phi)*sq_vert(theta),
>    sq_horz(phi) ]';
> plot3d( %, theta=0..1, phi=0..1/2 );
>
```

Exercise: Draw the cube with its top and bottom faces removed. Can you remove any other face?

```
>
```

In the section on the parameterization of the sphere we used animations to visualize how the parameterization could be "unfolded" in the direction of each parameter. Here, as a reminder, is the animation of the sphere parameterization being unfolded in the θ direction, around the horizontal circles. This animation is followed by the analogous animation for the cube parameterization. Look carefully at the two parameterizations (given just above each animation) and see how they are structured in the same way. Be sure that you understand how each animation works and why they look so similar.

```
> (theta,phi) -> [ sin(phi)*cos(theta),
>                  sin(phi)*sin(theta),
>                  cos(phi) ];
> plots[animate3d]( %(s*theta, phi), theta=0..2*Pi, phi=0..Pi,
>                   s=0..1, frames=60, orientation=[-60,60],
>                   title="Animated Sphere");
>
> (theta,phi) -> [ sq_vert(phi)*sq_horz(theta),
>                  sq_vert(phi)*sq_vert(theta),
>                  sq_horz(phi) ];
> plots[animate3d]( %(s*theta, phi), theta=0..1, phi=0..1/2,
>                   s=0..1, frames=60, orientation=[-60,60],
>                   title="Animated Cube");
>
```

Here are the animations of the sphere and cube parameterizations being unfolded in the other direction.

```
> (theta,phi) -> [ sin(phi)*cos(theta),
>                  sin(phi)*sin(theta),
>                  cos(phi) ];
> plots[animate3d]( %(theta, s*phi), theta=0..2*Pi, phi=0..Pi,
>                   s=0..1, frames=60, orientation=[30,120],
>                   title="Animated Sphere");
>
```

```
> (theta,phi) -> [ sq_vert(phi)*sq_horz(theta),
>                  sq_vert(phi)*sq_vert(theta),
>                  sq_horz(phi) ];
> plots[animate3d]( %(theta, s*phi), theta=0..1, phi=0..1/2,
>                   s=0..1, frames=60, orientation=[30,120],
>                   title="Animated Cube");
>
```

Exercise: The following two animations make use of a slight variation on the cube parameterization. Each animation unfolds this parameterization in a different direction. Explain what has been changed from the previous cube parameterization and why the changes have the effect that they do on the parameterization's unfoldings. (Be sure to rotate these animations as they are playing and watch them from various angles.)

```
> (theta,phi) -> [ sq_horz(phi)*sq_horz(theta),
>                  sq_horz(phi)*sq_vert(theta),
>                  sq_vert(phi) ];
> plots[animate3d]( %(theta, s*phi), theta=0..1/2, phi=0..1,
>                   s=0..1, frames=60, orientation=[30,60],
>                   axes=boxed, title="Animated Cube");
>
> (theta,phi) -> [ sq_horz(phi)*sq_horz(theta),
>                  sq_horz(phi)*sq_vert(theta),
>                  sq_vert(phi) ];
> plots[animate3d]( %(s*theta, phi), theta=0..1/2, phi=0..1,
>                   s=0..1, frames=60, orientation=[120,60],
>                   axes=framed, title="Animated Cube");
>
```

Here is a parameterization of a tetrahedron, a surface with a triangular profile and triangular cross sections.

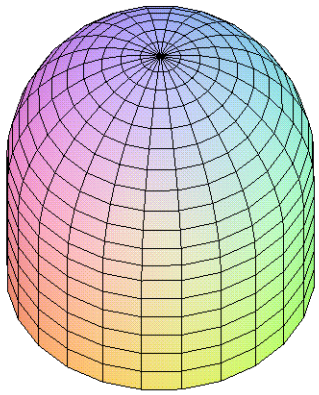
```
> '[ tri_vert(phi)*tri_horz(theta),
>    tri_vert(phi)*tri_vert(theta),
>    tri_horz(phi) ]';
> plot3d( %, theta=0..1, phi=0..1/2 );
>
```

There is a lot of experimenting and playing around that can be done with these ideas. For example, here is a surface created from an odd mix of component functions from three different parametric curves (circle, square, and triangle).

```
> '[sin(2*Pi*phi)*cos(2*Pi*theta),
>   sin(2*Pi*phi)*sq_vert(theta),
>   tri_horz(phi)]';
> plot3d( %, theta=0..1, phi=0..1/2 );
```

```
>
```

Exercise: Find a parameterization for the following surface. (Note: Do note just glue together half of a sphere with half of a cylinder using the **display** command. Find component functions that completely parameterize this surface.)



```
>
```

In all of the surfaces that we have parameterized so far, the vertical profile has been a closed curve. But it need not be. For example, here is a surface with triangular cross sections and a parabolic profile.

```
> '[(1+phi^2)*tri_horz(theta), (1+phi^2)*tri_vert(theta),
  phi]';
> plot3d(%, theta=0..1, phi=-1..1, scaling=constrained );
```

Notice in this parameterization that the curve that creates the vertical profile is the graph of a function with the independent variable on the z-axis. This vertical profile function ($g(u) = 1 + u^2$ in this case) is parameterized by the ϕ along the z-axis and the $1 + \phi^2$ term in the other two component functions (compare this to the way in which we parametrically graphed $x = f(y)$ in Section 5.4.2).

```
>
```

Here is a similar example with square cross sections and a sinusoidal profile.

```
> '[(2+cos(phi))*sq_horz(theta), (2+cos(phi))*sq_vert(theta),
  phi]';
> plot3d(%, theta=0..1, phi=-2*Pi..2*Pi, scaling=constrained );
>
```

Exercise: Explain how the last two examples are related to the idea, from calculus, of a "surface of revolution".

```
>
```

Now let us do some examples with the standard parameterization of the torus. Here again is the parameterization.

```
> [ (2+cos(phi))*cos(theta),
>   (2+cos(phi))*sin(theta),
>   sin(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..2*Pi, scaling=constrained );
```

Recall that a torus can be thought of as a "circle's worth of vertical circles". The $\cos(\phi)$ and $\sin(\phi)$ terms parameterize the vertical circles, and the $\cos(\theta)$ and $\sin(\theta)$ terms parameterize the circle of centers for the vertical circles. We can replace these functions with the component functions from other parametric curves and get other kinds of "tori".

```
>
```

Here is the parameterization of the surface that can be described as a "circle's worth of vertical trinagles".

```
> [ (2+tri_horz(theta))*cos(phi),
>   (2+tri_horz(theta))*sin(phi),
>   tri_vert(theta) ]:
> plot3d( %, theta=0..1, phi=0..11*Pi/6, scaling=constrained );
>
```

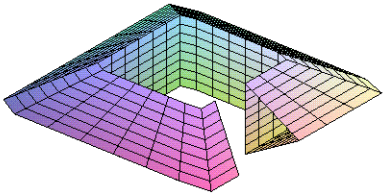
Here is the parameterization of the surface that can be described as a "square's worth of vertical circles".

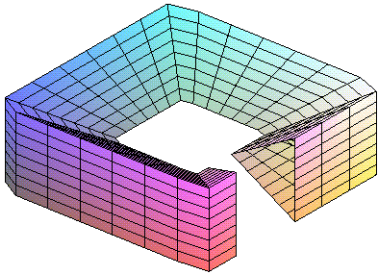
```
> [ (2+cos(theta))*sq_horz(phi),
>   (2+cos(theta))*sq_vert(phi),
>   sin(theta) ]:
> plot3d( %, theta=0..2*Pi, phi=0..11/12, scaling=constrained
  );
>
```

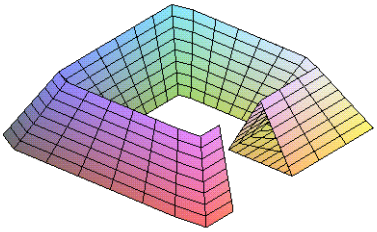
Here is the surface that can be described as a "triangle's worth of vertical squares".

```
> [ (2+sq_horz(theta))*tri_horz(phi),
>   (2+sq_horz(theta))*tri_vert(phi),
>   sq_vert(theta) ]:
> plot3d( %, theta=0..1, phi=0..11/12, scaling=constrained );
>
```

Exercise: Each of the following three surfaces is a torus that can be thought of as a "square's worth of vertical triangles". Find a parameterization for each of these three surfaces.







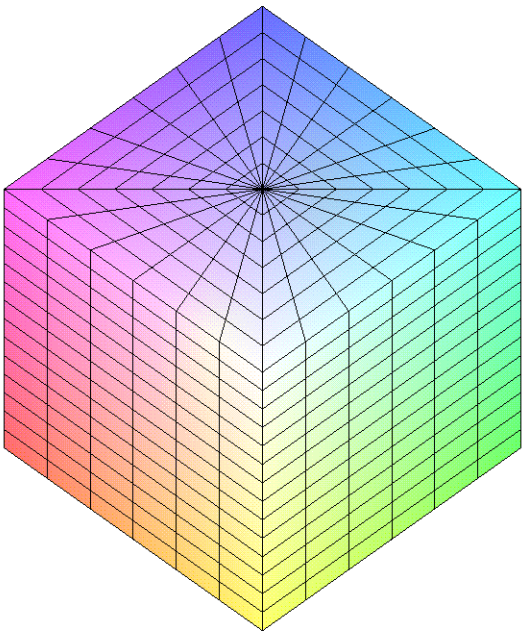
```
>
```

Exercise: The following three parametric surfaces are all based on the parameterization of a torus. Analyze them to determine which part of the parameterization lifts the torus up in the z-direction. Which part of the parameterization makes the vertical cross section shrink as it winds around? Which part of the parameterization makes the torus spiral in towards the z-axis?

```
> [ (1-theta/2.5)*(5+sq_horz(phi))*cos(2*Pi*theta),
>   (1-theta/2.5)*(5+sq_horz(phi))*sin(2*Pi*theta),
>   2*theta+(1-theta/2.5)*sq_vert(phi) ]:
> plot3d( %, theta=0..2.5, phi=0..1,
>         orientation=[-30,75], numpoints=1000 );
>
> [ (2+(1-theta/5)*sq_horz(phi))*cos(2*Pi*theta),
>   (2+(1-theta/5)*sq_horz(phi))*sin(2*Pi*theta),
```

```
>   2*theta+(1-theta/5)*sq_vert(phi) ]:
> plot3d( %, theta=0..2.5, phi=1/2..1,
>           orientation=[-30,75], numpoints=1000 );
>
> [ (2+(1-theta/2)*sq_horz(phi))*cos(2*Pi*theta),
>   (2+(1-theta/2)*sq_horz(phi))*sin(2*Pi*theta),
>   (1-theta/2)*sq_vert(phi) ]:
> plot3d( %, theta=0..1.5, phi=1/2..1,
>           orientation=[-60,45], numpoints=1000 );
>
```

Exercise: Use homotopies to recreate the following animation of a cube morphing into a sphere. (See Section 4.10 and also Section 5.4.4.)



```
>

>
```

5.7.5. Exercises with parametric surfaces

Exercise: Explain in detail how each of the following three parameterizations defines its parametric surface.

```
> plot3d( [ u*cos(v), u*sin(v), 1 ], u=0..1, v=0..2*Pi );
> plot3d( [ u*cos(v), u*sin(v), u ], u=0..1, v=0..2*Pi );
> plot3d( [ u*cos(v), u*sin(v), v ], u=0..1, v=0..2*Pi );
```

```
>
```

Exercise: Study the following parameterization. How would you get more spirals to show in the graph? What if you wanted this to be a spiral water trough with no top?

```
> plot3d( [(2+sin(v))*cos(u),
>          (2+sin(v))*sin(u),
>          u+cos(v)], u=0..4*Pi, v=0..2*Pi );
>
```

Exercise: Can you make this one into an open topped water trough? How can you get more spirals? (Hint: Experiment with the various constants in the formulas and try to get a sense of what they determine.)

```
> [ (1-u)*(3+cos(v))*cos(4*Pi*u),
>   (1-u)*(3+cos(v))*sin(4*Pi*u),
>   3*u+(1-u)*sin(v) ];
> plot3d( %, u=0..1, v=0..2*Pi, orientation=[-14,76] );
>
```

Exercise: Part (a): Let f(x) be a real valued function of one variable. Find a way to parameterize the surface of revolution generated by revolving the graph of f(x) around the x-axis. Graph a few surfaces of revolution for different functions f.

```
>
```

Part (b): Find a way to parameterize the surface of revolution generated by revolving the graph of f around the y-axis. Around the z-axis.

```
>
```

Exercise: Part (a) Here are four different parameterizations of the sphere of radius one centered at the origin. For each parameterization try to change just one number from one range so that the parameterization draws only the upper hemisphere. If you cannot do this by changing just one number, explain why.

```
> [ sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi, title="Sphere" );

> [ cos(phi)*cos(theta), cos(phi)*sin(theta), sin(phi) ];
> plot3d( %, theta=0..Pi, phi=0..2*Pi, title="Sphere" );

> [ sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..Pi, phi=0..2*Pi, title="Sphere" );

> [ sqrt(1-z^2)*cos(theta), sqrt(1-z^2)*sin(theta), z ];
> plot3d( %, theta=0..2*Pi, z=-1..1, title="Sphere" );
```

Part (b) Repeat part (a) but this time try to draw only the right hemisphere.

```
```

```
> 
```

Part (c) Repeat part (a) but this time try to draw 3/4 of the sphere (say, the upper hemisphere and the left or right half of the lower hemisphere).

```
> 
```

Exercise: This exercise compares the second of the four sphere parameterizations above with a parameterization of a torus. Explain how the following minor change in the sphere parameterization manages to become half of a torus. First, the sphere parameterization again.

```
> [ cos(phi)*cos(theta), cos(phi)*sin(theta), sin(phi) ];
> plot3d( %, theta=0..Pi, phi=0..2*Pi, title="Sphere" );
```

Now a minor change that makes it into a parameterization of a torus.

```
> [ (2+cos(phi))*cos(theta), (2+cos(phi))*sin(theta), sin(phi)
  ];
> plot3d( %, theta=0..Pi, phi=0..2*Pi, title="1/2 Torus" );
> 
```

Exercise: Compare the following two parameterizations of a sphere of radius one centered at the origin. In what way do they differ? In what way are they similar?

```
> [ sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi, title="Sphere" );
> 
> [ sin(phi)*cos(theta), cos(phi), sin(phi)*sin(theta) ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi, title="Sphere" );
> 
```

Exercise: Part (a): Recall that in the previous section of this worksheet we gave examples of drawing curves on surfaces. Let us draw some curves on parametric surfaces. Here is the standard parameterization of the sphere written as a Maple function and with the domain of the parameterization being the unit square.

```
> h := (u,v) -> [ sin(Pi*v)*cos(2*Pi*u), sin(Pi*v)*sin(2*Pi*u),
  cos(Pi*v) ];
```

The next command graphs the sphere and then gives the graph a name for later use.

```
> plot3d( h(u,v), u=0..1, v=0..1, style=patchnogrid );
> graph1 := %:
```

The next function parameterizes a half circle contained in the unit square.

```
> g := t -> ( (cos(2*Pi*t)+1)/2, sin(2*Pi*t)/2+1 );
> plot( [ g(t), t=1/2..1 ], view=[0..1,0..1] );
```

The next graph uses the parameterization of the sphere to "push" this parametric curve onto the graph of the parametric surface.

```
> plots[spacecurve]( h(g(t)), t=1/2..1, color=black );
> graph2 := %:
```

Now draw the curve on the surface with the surface.

```
> plots[display](graph1, graph2);
```

Try to explain the shape of the resulting curve on the surface. For example, why is it closed? (Hint: Think carefully about what the **u** and **v** coordinates from the unit square do in the parametric surface.)

Part (b): Here is a similar example. The next function parameterizes a different half circle contained in the unit square.

```
> g := t -> ( cos(2*Pi*t)/2+1, (sin(2*Pi*t)+1)/2 );
> plot( [ g(t), t=1/4..3/4 ], scaling=constrained,
  view=[0..1,0..1] );
```

The next graph uses the parameterization of the sphere to "push" this parametric curve onto the graph of the parametric surface.

```
> graph2 := plots[spacecurve]( h(g(t)), t=1/4..3/4, color=black
  ):
```

Now draw the curve on the surface with the surface.

```
> plots[display](graph1, graph2);
```

Try to explain the shape of the resulting curve on the surface. For example, why is this curve not closed?

Part (c): One more example. The next function parameterizes a whole circle contained in the unit square.

```
> g := t -> ( (cos(2*Pi*t)+1)/2, (sin(2*Pi*t)+1)/2 );
> plot( [ g(t), t=0..1 ], scaling=constrained, view=[0..1,0..1]
  );
```

The next graph uses the parameterization of the sphere to "push" this parametric curve onto the graph of the parametric surface.

```
> graph2 := plots[spacecurve]( h(g(t)), t=0..1, color=black ):
```

Now draw the curve on the surface with the surface.

```
> plots[display](graph1, graph2);
```

Try to explain the shape of the resulting curve on the surface. For example, why is this curve closed with a figure eight like shape?

```
>
```

Exercise: Given a non vertical plane in three dimensional space and a point on the plane, draw a square centered at the point and lying in the plane. Also draw the projection of the square onto the *xy*-plane.

```
>
```

Exercise: Part (a) Given a plane through the origin, draw a disk centered at the origin and lying in the plane.

```
>
```

Part (b) Given a non vertical plane and a point on the plane, draw a disk centered at the point and lying in the plane and draw the projection of the disk onto the *xy*-plane.

```
>
```

```
> 
```

5.8. Non Cartesian coordinate systems in space

Just as the **plot** command can use non Cartesian coordinates on the plane when it graphs real valued functions and parametric curves, the **plot3d** command can use non Cartesian coordinates on three dimensional space when it graphs real valued functions of two variables and parametric surfaces. Here is an example of the same function being graphed using three different coordinate systems in space, Cartesian, cylindrical, and spherical.

```
> plot3d( x^2+y^2, x=-2..2, y=-2..2 );
> plot3d( x^2+y^2, x=-2..2, y=-2..2, coords=cylindrical );
> plot3d( x^2+y^2, x=-2..2, y=-2..2, coords=spherical );
>
```

The following two subsections give the details of using **plot3d** to graph real valued functions of two variables using first the cylindrical coordinate system and then the spherical coordinate system. The next subsection returns to the idea of graphing curves on surfaces. The subsection after that shows how to use **plot3d** to graph parametric surfaces using non Cartesian coordinates. The last subsection explains how parametric surfaces can be used to get around a bug in how Maple graphs functions over "non rectangular domains" when using non Cartesian coordinates.

```
>
```

5.8.1. Real valued functions and cylindrical coordinates

Just as the **plot** command must, for each coordinate system on the plane, (arbitrarily) choose one coordinate direction for the independent variable of the function, the **plot3d** command must, for each coordinate system on three dimensional space, (arbitrarily) choose two coordinate directions for the independent variables of the function. In addition, the **plot3d** command must choose an ordering for the two independent variables, that is, a way to match up each of the two ranges in the **plot3d** command with one of the two preferred coordinate directions. We have already seen how **plot3d** does this for Cartesian coordinates in space. Now we want to look at how **plot3d** makes choices for the cylindrical coordinate system.

Let us use the common labels θ, r, and z for the coordinates in the cylindrical coordinate system. When graphing a function in cylindrical coordinates, the **plot3d** command chooses the angular and vertical directions (i.e., θ and z) as the independent variables and the radial direction (i.e., r) as the dependent variable. In addition, the first range in the **plot3d** command will be associated to the radial direction and the second range will be associated to the vertical direction. In other words, given a function f of two variables, the **plot3d** command with cylindrical coordinates will draw the graph of $r = f(\theta, z)$. So for example, we graph a cylinder by graphing a constant function.

```
> plot3d( 3, theta=0..2*Pi, z=-6..6, coords=cylindrical,
  axes=boxed );
```

The equation $r = \sin(3\,\theta)$ in polar coordinates graphs a three petal rose in the plane. Here is a "cylinder" over this three petal rose.

```
> plot3d( sin(3*t), t=0..2*Pi, z=0..1/2, coords=cylindrical,
  axes=boxed );
```

The above graph does not look very good. We can improve it by using another option. (We will see in the next worksheet what this option does and why it is needed.)

```
> r := sin(3*theta);
> plot3d( r, theta=0..2*Pi, z=0..1/2, coords=cylindrical,
  axes=boxed,
          grid=[40,40] );
```

Recall that the equation $r = a\,(1 + 2\cos(\theta))$ in polar coordinates defines a limacon in the plane with a "diameter" determined by a. Here is an example with $a = 5$.

```
> plot( 5*(1+2*cos(t)), t=0..2*Pi, coords=polar,
  scaling=constrained );
```

If we graph the function $f(\theta, z) = 5\,(1 + 2\cos(\theta))$ as a function of two variables with cylindrical coordinates, the graph will be a "cylinder" over the above limacon.

```
> r := 5*(1+2*cos(theta));
> plot3d( r, theta=0..2*Pi, z=-1..1, coords=cylindrical,
>         axes=boxed, grid=[40,40] );
```

Now let the "diameter" of the limacon vary with z. In the next graph, every horizontal cross section is a limacon, but the "diameters" depends on z.

```
> r := (1+z^2)*(1+2*cos(theta));
> plot3d( r, theta=0..2*Pi, z=-1..1, coords=cylindrical,
>         axes=boxed, grid=[40,40] );
>
```

Given a function of one variable, it is easy to use cylindrical coordinates to graph its surface of revolution around the z-axis. Here is a simple example.

```
> sqrt(z);
> plot3d( %, theta=0..2*Pi, z=0..4, coords=cylindrical );
```

Here is an animation of the one dimensional graph being revolved around the z-axis to create the surface of revolution.

```
> p := t -> plot3d( sqrt(z), theta=0..t, z=0..4,
  coords=cylindrical ):
> seq( p(2*Pi*i/50), i=1..50 ):
> plots[display]( [%], insequence=true );
```

Here are a few other surfaces of revolution.

```
> exp(-z^2);
> plot3d( %, theta=0..15*Pi/8, z=-2..2, coords=cylindrical );
> z+sin(z);
> plot3d( %, theta=0..2*Pi, z=0..8*Pi, coords=cylindrical );
```

```
> 1+sin(z)/z;
> plot3d( %, theta=0..2*Pi, z=-3*Pi..3*Pi, coords=cylindrical
  );
```

Here is the last function again, but with a different range. This example shows that occasionally the **plot3d** command can draw misleading graphs. The following surface should not be broken into two pieces.

```
> 1+sin(z)/z;
> plot3d( %, theta=0..2*Pi, z=-4*Pi..4*Pi, coords=cylindrical
  );
```

Here is a torus (a donut shaped surface) drawn as a surface of revolution for two functions. (Try graphing each of the two surfaces by themselves.)

```
> 2+sqrt(1-z^2), 2-sqrt(1-z^2);
> plot3d( {%}, theta=0..2*Pi, z=-1..1, coords=cylindrical );
>
```

Exercise: Part (a) Use **plot3d** with cylindrical coordinates to graph a sphere of radius 4 centered at the origin.

```
>
```

Part (b) Now drill a cylindrical hole of radius three through the sphere along the *z*-axis. Draw the remaining part of the sphere along with the wall of the cylindrical hole.

```
>
```

Part (c) Cut away part of the graph from part (b) so that you can see the space between the wall of the sphere and the wall of the hole. Make both a horizontal and a vertical cutaway.

```
>
```

Exercise: In Section 5.7.5 there is an exercise asking you to figure out how to use parametric surfaces in rectangular coordinates to parameterize a surface of revolution around any one of the three axes. Compare your solution for the *z*-axis with the cylindrical coordinate technique given here.

```
>
```

Given a function f of two variables, there are a total of six ways that we could graph f in cylindrical coordinates. We could choose to graph $r = f(\theta, z)$, $r = f(z, \theta)$, $z = f(\theta, r)$, $z = f(r, \theta)$, $\theta = f(r, z)$, or $\theta = f(z, r)$. By default, the **plot3d** command will only draw the first of these six possible graphs. In the next section of this worksheet we will see how we can draw all six of these graphs by using parametric equations. And in the next worksheet we will see how we can draw all of these graphs by defining new versions of cylindrical coordinates.

Why is it that **plot3d** defaults to $r = f(\theta, z)$? Recall that in polar coordinates we usually graph functions of the form $r = f(\theta)$. And since cylindrical coordinates is just polar coordinates with the variable *z* added, it seems reasonable to just consider the *z* coordinate as another independent

variable and, by analogy to polar coordinates, graph functions of the form $r = f(\theta, z)$.

Of the six possible graphs that we could make in cylindrical coordinates, there is one other graph that would seem to be a very reasonable choice as the default graph for **plot3d**. Since the default graph in rectangular coordinates is of the form $z = f(x, y)$, and since (x, y) and (r, θ) both coordinatize the plane, then by analogy to rectangular coordinate it would seem reasonable for **plot3d** to graph $z = f(r, \theta)$. Such graphs can in fact be very useful. For example, suppose that in a calculus class we want to find the volume under the graph of a function $f(x, y)$ and over the cardioid defined by $r = 1 + \cos(\theta)$, and we want to visualize this volume before computing it. So we need to draw a graph of $f(x, y)$ over the cardioid $r = 1 + \cos(\theta)$. We might try to do this using **plot3d**'s ability to graph over non rectangular domains in rectangular coordinates, but that would be difficult. What we would like to do is convert the function to cylindrical coordinates using $g(r, \theta) = f(r\cos(\theta), r\sin(\theta))$ and then draw a graph of $z = g(r, \theta)$ using cylindrical coordinates with the variable θ ranging between 0 and 2π and the variable r ranging between 0 and $1 + \cos(\theta)$. But **plot3d** cannot graph a function of the form $z = g(r, \theta)$, so we will come back to this example in Section 5.8.4.

```
> 
```

In Section 5.6.4 we looked at graphing curves on surfaces. Let us try graphing a curve on the surface of a sphere. Since a sphere is a surface of revolution, we can use cylindrical coordinates to draw a sphere as the graph of a function of two variables. Here is the function of two variables.

```
> f := (theta, z) -> sqrt(1-z^2);
```

The domain for the graph of a sphere is θ between 0 and 2π and z between -1 and 1.

```
> plot3d( f(theta, z), theta=0..2*Pi, z=-1..1,
  coords=cylindrical,
>                        style=patchnogrid );
```

Let us give this graph a name for later use.

```
> g1 := %:
```

Here is a simple curve in the domain of our function, a half circle.

```
> h := t -> (cos(t)+Pi, sin(t)+1);
> plot( [h(t), t=Pi..2*Pi], view=[0..2*Pi,-1..1],
  scaling=constrained );
```

Now use the function **f** to "lift" the curve **h** onto the graph of the sphere.

```
> g2 := plots[spacecurve](
>        [ f(h(t)), h(t) ],
>        t=Pi..2*Pi, coords=cylindrical, color=black,
  numpoints=100 ):
```

(If you compare the last **spacecurve** command to the equivalent one from Section 5.6.4, you will notice a subtle difference. We will explain the reason for this difference in Section 5.8.4.)

Now combine the sphere with the curve on the sphere.

```
> plots[display](g1, g2);
```

Can you explain the shape of this curve? For example, why is it closed?

```
>
```

Here is a slightly different half circle in the domain of our function **f**.

```
> h := t -> (2*cos(t)+Pi, 2*sin(t)+1);
> plot( [h(t), t=Pi..2*Pi], view=[0..2*Pi,-1..1],
  scaling=constrained );
```

Use **f** to "lift" the curve **g** onto the graph of the sphere.

```
> g2 := plots[spacecurve](
>       [ f(h(t)), h(t) ],
>       t=Pi..2*Pi, coords=cylindrical, color=black,
  numpoints=300 ):
> plots[display](g1, g2);
>
```

Exercise: Notice that the main difference between the last two examples is the radius of the circle in the domain of **f**. The first example has radius 1 and the second example has radius 2. Create an animation that shows the first example morphing into the second example.

```
>
```

Exercise: Here is a different half circle in the domain of our function **f**.

```
> h := t -> (cos(t)+2*Pi, sin(t));
> plot( [h(t), t=Pi/2..3*Pi/2], view=[0..2*Pi,-1..1],
  scaling=constrained );
```

Use **f** to "lift" the curve **g** onto the graph of the sphere.

```
> g2 := plots[spacecurve]( [f(h(t)), h(t)], t=Pi/2..3*Pi/2,
  coords=cylindrical, color=black ):
> plots[display](g1, g2);
```

Why is this curve not closed?

```
>
```

```
>
```

5.8.2. Real valued functions and spherical coordinates

Now let us turn to spherical coordinates. Let us use the common labels ρ, θ, and ϕ to represent the coordinates. When graphing a function of two variables in spherical coordinates, the **plot3d** command uses θ and ϕ as the independent variables. The first range in the **plot3d** command will be associated to θ and the second range will be associated to ϕ. So given a function f of two variables, the **plot3d** command with spherical coordinates will draw the graph of $\rho = f(\theta, \phi)$. For example, we can graph a sphere using spherical coordinates by graphing a constant function. The following command graphs a sphere but with a bit of its top removed and with a vertical slice taken out. Try closing each of these holes (one at a time).

```
```

```
> plot3d( 1, t=0..7*Pi/4, p=Pi/8..Pi, coords=spherical );
>
```

Exercise: Here is a graph of a function of two variables in spherical coordinates. This is a "bumpy sphere" with a hole in the bottom.

```
> r := 1+.2*cos(5*theta)*cos(5*phi);
> plot3d( r, theta=0..2*Pi, phi=0..3*Pi/4, coords=spherical );
```

How would you change the size of the hole? How would you cut away the front half of the graph? How would you make the bumps bigger or smaller? More or less numerous? Make a 3D animation of the hole growing and shrinking. Make another animation of a "pulsating sphere" with the bumps growing and shrinking.

```
>
```

```
>
```

5.8.3. More curves on surfaces

In previous sections we saw how we can combine parametric curves with graphs of surfaces. Let us do some examples using non Cartesian coordinate systems. Here is an example of a curve drawn in cylindrical coordinates on a surface drawn in rectangular coordinates.

```
> f := (x,y) -> x^2-y^2;
> graph1 := plot3d( f(x,y), x=-1..1,
  y=-sqrt(x^2+1)..sqrt(x^2+1),
>                    style=patchnogrid ):
> r := t -> sin(4*t);
> graph2 := plots[spacecurve]( [r(t), t,
  f(r(t)*cos(t),r(t)*sin(t))],
>           t=0..2*Pi, coords=cylindrical, color=black,
  numpoints=200 ):
> plots[display](graph1, graph2);
> r := 'r':
>
```

Exercise: Change the curve drawn on the surface from a rose to the cardioid $r = 1 + 2\cos(\theta)$.

```
>
```

Here is an example of a curve drawn in rectangular coordinates on a surface drawn with cylindrical coordinates (a sine curve running up and down the side of a cylinder).

```
> g1 := plot3d( 1, theta=0..2*Pi, z=0..3*Pi,
  coords=cylindrical,
>                style=hidden, shading=xy ):
> g2 := plots[spacecurve]( [2/3*sin(2*t),
  sqrt(1-(2/3*sin(2*t))^2), t],
>                            t=0..3*Pi, color=black,
```

```
   numpoints=100 ):
> plots[display]( g1, g2 );
>
```

Exercise: Modify the last example so that the sine curve lies in the *yz*-plane inside the cylinder and then make the cylinder "see through" so that you can see the sine curve inside the cylinder.

```
>
```

Exercise: Part (a): Modify the sine curve on a cylinder example so that the cylinder has a diameter that depends on *z* (try 2 + *z* as the expression for the diameter). Make sure that the graph of the sine curve stays on the graph of the new surface.

```
>
```

Part (b): Now modify the graph from part (a) so that the amplitude of the sine curve also increases as it moves up the graph in the *z* direction (and again, keep the graph of the new sine curve on the graph of the surface).

```
>
```

Exercise: Take any one of the above examples of a curve drawn on a surface and convert the curve into a (pretty narrow) tube plot. Try this both with and without the surface itself in the graph.

```
>
```

```
>
```

5.8.4. Parametric surfaces in non Cartesian coordinates

The **plot3d** command can draw parametric surfaces using other spatial coordinates systems besides rectangular coordinates. Here is an example that uses cylindrical coordinates. This is a ribbon winding its way up the side of a paraboloid.

```
> plot3d( [sqrt(theta+t), theta, theta+t], theta=0..7*Pi,
  t=0..3,
>          coords=cylindrical, style=patchnogrid, grid=[50,50]
  );
```

Here is a similar example on the surface of a sphere and done with spherical coordinates. It looks like an apple being peeled.

```
> plot3d( [1, 8*t+s, t], t=0..Pi, s=-2..2,
>          coords=spherical, style=patchnogrid, grid=[50,50] );
```

Here is a sphere with its surface being peeled off of it.

```
> g1:=plot3d( [1+.2*(Pi-t), 8*t+s, t], t=0..Pi, s=-2.4..2.4,
>             coords=spherical, style=patchnogrid, grid=[50,50]
  ):
> g2:=plot3d( 1, theta=0..2*Pi, phi=0..Pi,
>             coords=spherical, style=hidden, grid=[50,50] ):
> plots[display]( g1, g2, scaling=constrained );
```

```
>
```

In Section 5.8.1 we mentioned that the **plot3d** command, when using cylindrical coordinates to graph a real valued function of two variables, gives the θ and z coordinates the preferred status of being the independent variables and **plot3d** always draws a graph of $r = f(\theta, z)$. When graphing a parametric surface in cylindrical coordinates, **plot3d** does not give any coordinate direction a preferred status. It does however always treat the first expression after the opening bracket as the r-component, the second expression as the θ-component, and the third expression as the z-component. As you might expect by now, given any function of two variables, we can use parametric equations to draw any of the six graphs $r = f(\theta, z)$, $r = f(z, \theta)$, $\theta = f(r, z)$, $\theta = f(z, r)$, $z = f(\theta, r)$, or $z = f(r, \theta)$. For example, here are the six different graphs of a constant function (over a non rectangular domain) in cylindrical coordinates.

```
> plot3d([1, theta, z], theta=0..2*Pi, z=0..theta,
  coords=cylindrical);
> plot3d([1, theta, z], theta=0..z,    z=0..2*Pi,
  coords=cylindrical);
> plot3d([r, 1, z],     r=0..2*Pi,     z=0..r,
  coords=cylindrical);
> plot3d([r, 1, z],     r=0..z,        z=0..2*Pi,
  coords=cylindrical);
> plot3d([r, theta, 1], theta=0..2*Pi, r=0..theta,
  coords=cylindrical);
> plot3d([r, theta, 1], theta=0..r,    r=0..2*Pi,
  coords=cylindrical);
>
```

Exercise: Draw the six different graphs in cylindrical coordinates of the function $f(u, v) = u^2 - v^2$, with the domain u between -1 and 1 and v between 2 and 3. For each graph, make sure that the graph makes sense for you.

```
>
```

In Section 5.8.1 we mentioned the problem of visualizing the graph of a function $z = f(x, y)$ over the cardioid $r = 1 + \cos(\theta)$. One way to do this is to use $g(r, \theta) = f(r\cos(\theta), r\sin(\theta))$ to convert the function to cylindrical coordinates and then use cylindrical coordinates to graph $z = g(r, \theta)$ with the variable θ ranging between 0 and 2 π and the variable r ranging between 0 and $1 + \cos(\theta)$. The problem with this idea is that the **plot3d** command with cylindrical coordinates will only graph functions of the form $r = g(\theta, z)$. We get around this limitation of **plot3d**, and get the graph that we want, by using parametric equations in cylindrical coordinates to graph $z = g(r, \theta)$. Here is an example with $f(x, y) = x^2 + y^2$, so $g(r, \theta) = r^2$.

```
> plot3d( [r,t,r^2], r=0..1+cos(t), t=0..2*Pi,
  coords=cylindrical );
```

The next graph shows that the previous graph is correct. The next graph redraws the previous graph and combines it with a graph of the paraboloid. From the next graph we see that the previous graph really is part of the paraboloid.

```
> g1:=plot3d( [r,t,r^2], r=0..1+cos(t), t=0..2*Pi,
  coords=cylindrical ):
> g2:=plot3d( x^2+y^2, x=-2..2, y=-sqrt(4-x^2)..sqrt(4-x^2),
>             style=wireframe ):
> plots[display]( g1, g2 );
```

Here is a way to add the "walls" to the volume under the graph of $f(x, y) = x^2 + y^2$ and over the cardioid $r = 1 + \cos(t)$, so that we can see the exact shape of this volume.

```
> g1:=plot3d( [r,t,r^2], r=0..1+cos(t), t=0..2*Pi,
  coords=cylindrical ):
> g2:=plot3d( [1+cos(t), t, (1+cos(t))^2*z], t=0..2*Pi, z=0..1,
>             coords=cylindrical ):
> plots[display]( g1, g2 );
>
```

Exercise: The graph named **g2** in the last example draws the walls of the volume. The walls are drawn as a parametric surface in cylindrical coordinates. Redraw the walls of the volume as the graph of a function $r = h(\theta, z)$ in cylindrical coordinates.

```
>
```

Exercise: Draw a graph of the volume under the function $f(x, y) = x + y$ and over petal in the first quadrant of the three leaf rose $r = \sin(3\,\theta)$. Be sure to draw both the top of the volume and its side walls.

```
>
```

In Section 5.8.2 we mentioned that the **plot3d** command, when using spherical coordinates to graph a function of two variables, gives the θ and ϕ coordinates the preferred status of being the independent variables and **plot3d** always draws a graph of $\rho = f(\theta, \phi)$. When graphing a parametric surface in spherical coordinates, **plot3d** does not give any coordinate direction a preferred status. It does however always treat the first expression after the opening bracket as the ρ-component, the second expression as the θ-component, and the third expression as the ϕ-component. And of course, given any function of two variables, we can use parametric equations to draw any of the six graphs $\rho = f(\theta, \phi)$, $\rho = f(\phi, \theta)$, $\theta = f(\rho, \phi)$, $\theta = f(\phi, \rho)$, $\phi = f(\theta, \rho)$, or $\phi = f(\rho, \theta)$. For example, here are the six different graphs of a constant function (over a non rectangular domain) in spherical coordinates.

```
> plot3d([1, theta, phi], theta=0..2*Pi, phi=0..theta,
  coords=spherical);
> plot3d([1, theta, phi], theta=0..phi,  phi=0..2*Pi,
  coords=spherical);
> plot3d([rho, 1, phi],   rho=0..2*Pi,   phi=0..rho,
```

```
   coords=spherical);
> plot3d([rho, 1, phi],   rho=0..phi,    phi=0..2*Pi,
   coords=spherical);
> plot3d([rho, theta, 1], theta=0..2*Pi, rho=0..theta,
   coords=spherical);
> plot3d([rho, theta, 1], theta=0..rho,  rho=0..2*Pi,
   coords=spherical);
>
```

Exercise: Draw the six different graphs in spherical coordinates of the function $f(u, v) = u^2 - v^2$, with the domain u between 0 and $\pi/2$ and v between $\pi/2$ and π.

```
>
```

Exercise: Draw a curve on the surface of a sphere by parameterizing the sphere using spherical coordinates and then using the parameterization to "push" a curve in the domain of the parameterization onto the sphere.

```
>
```

```
>
```

5.8.5. Non rectangular regions: fixing a bug in Maple

Here is an example of how we can make use of parametric graphs of functions. There is a bug in Maple in the way that the **plot3d** command handles "non rectangular domains" in either cylindrical or spherical coordinates and we can get around this bug by graphing functions in their parametric form.

First an example of a non rectangular domain in cylindrical coordinates. The following command is supposed to graph $r = f(\theta, z)$ in cylindrical coordinates where f is the constant function 1. Notice that in this command, the range of the second variable (z) depends on first variable (θ). The following graph is not correct.

```
> plot3d( 1, theta=0..2*Pi, z=0..theta, coords=cylindrical );
```

Here is what the graph should have looked like. Notice that all we are doing here is converting the above graph of a function into its equivalent graph as a parametric surface, as explained in the last subsection. We can tell that the following graph is correct for a couple of reasons. First, in cylindrical coordinates the graph of $r = f(\theta, z)$, where f is a constant function, should be a cylinder. Second, notice how in the next graph, the range for the z variable gets larger as we go around the θ direction. This is as it should be, since the "non rectangular" domain has the range for z depending on θ.

```
> plot3d( [1, theta, z], theta=0..2*Pi, z=0..theta,
   coords=cylindrical );
```

Here is how we can reproduce the buggy graph above as a parametric surface. The next command is drawing a graph of the form $z = f(r, \theta)$ but the radial variable is called **theta** in

the next command, the angular variable is called **z**, and the function f is the constant function **1**.

```
> plot3d( [theta, z, 1], theta=0..2*Pi, z=0..theta,
  coords=cylindrical );
```

So now we see that the **plot3d** command somehow gets coordinate directions mixed up when we try to use non rectangular domains with the graph of a function in cylindrical coordinates.

```
>
```

Now here is an example using a "non rectangular" domain in spherical coordinates. This graph is not correct.

```
> plot3d( 1, t=0..2*Pi, p=Pi/4+.2*sin(5*t)..Pi,
  coords=spherical );
```

Here is what the graph should have looked like. Notice that all we are doing here is converting the above graph of a function into its equivalent graph as a parametric surface.

```
> plot3d( [1, t, p], t=0..2*Pi, p=Pi/4+.2*sin(5*t)..Pi,
  coords=spherical );
```

Exercise: Explain how we can tell that the above graph is correct.

```
>
```

Here is how we can reproduce the buggy graph as a parametric surface. The next command is drawing a graph of the form $\phi = f(\rho, \theta)$ but the radial variable is called **t** and the angular variable is called **p** and the function is the constant function **1** (so ϕ is constantly equal to 1 in the graph, which explains the cone like slope of the surface).

```
> plot3d( [t, p, 1], t=0..2*Pi, p=Pi/4+.2*sin(5*t)..Pi,
  coords=spherical );
```

As in the case of cylindrical coordinates, **plot3d** somehow gets coordinate directions mixed up when we try to use non rectangular domains with the graph of a function in spherical coordinates.

```
>
```

Exercise: There is a similar bug in the **animatecurve** command. Let us look at an example that brings out this bug. Here is a graph of a function.

```
> plot( sin(4*x), x=0..2*Pi );
```

Let us animate this last graph.

```
> plots[animatecurve]( sin(4*x), x=0..2*Pi );
```

Now let us convert the graph of the function from Cartesian to polar coordinates.

```
> plot( sin(4*x), x=0..2*Pi, coords=polar );
```

Now let us convert the animation from Cartesian to polar coordinates (which should animate this last graph).

```
> plots[animatecurve]( sin(4*x), x=0..2*Pi, coords=polar );
```

What went wrong? What did **animatecurve** do? Find a way to use **animatecurve** to animate the correct graph in polar coordinates.

```
>
```

Exercise: The name that Maple gives to Cartesian coordinates in three dimensional space is `rectangular`. This is the default coordinate system for the `plot3d` command. So why do the following two `plot3d` commands produce different graphs? Explain what goes wrong in the second graph. Use parametric surfaces to justify your explanation.

```
> f := (x,y) -> x^2+y^2;
> plot3d( f, -2..2, -2..2, axes=boxed );
> plot3d( f, -2..2, -2..2, axes=boxed, coords=rectangular );
>
```

```
>
```

5.9. Graphs of equations

Maple can draw graphs of equations in two or three variables, it can graph equations in several coordinate systems, and it can graph more than one equation at a time. For example, here is how we can draw "graph paper" for Cartesian coordinates.

```
> plots[implicitplot]( {x=-2, x=-1, x=0, x=1, x=2,
                        y=-2, y=-1, y=0, y=1, y=2},
                        x=-3..3, y=-3..3, axes=framed );
```

And here is some "graph paper" for polar coordinates.

```
> plots[implicitplot]( {r=0, r=1, r=2, r=3, r=4, theta=0,
  theta=Pi/4,
                        theta=Pi/2, theta=3*Pi/4, theta=Pi,
  theta=5*Pi/4,
                        theta=3*Pi/2, theta=7*Pi/4},
                        r=0..4, theta=0..2*Pi, axes=framed,
  coords=polar);
```

It does not do much good to draw "graph paper" in three dimensions (why?). Instead, here are graphs of three "coordinate planes" for each of the rectangular, cylindrical, and spherical coordinate systems.

```
> plots[implicitplot3d]( {x=0, y=0, z=0}, x=-1..1, y=-1..1,
  z=-1..1 );
> plots[implicitplot3d]( {r=1, theta=0, z=0},
                         r=0..2, theta=0..2*Pi, z=-2..2,
                         coords=cylindrical );
> plots[implicitplot3d]( {rho=1, theta=Pi, phi=Pi/4},
                         rho=0..1.5, theta=Pi/8..15*Pi/8,
  phi=0..Pi,
                         numpoints=800, coords=spherical );
>
```

Exercise: Explain why the following graph is not a circle

```
> plots[implicitplot]( r=1, theta=0..2*Pi, r=0..2,
>                      coords=polar, scaling=constrained );
```

Explain why the following graph is not a sphere and explain in detail exactly why it has the shape that it does.

```
> plots[implicitplot3d]( rho=1, theta=0..2*Pi, phi=0..Pi,
  rho=0..1,
>                        coords=spherical, scaling=constrained );
>
```

Exercise: Explain in detail why the following graph has the shape that it does.

```
> plots[implicitplot3d]( 1-u=sin(w)/2, u=0..2, v=0..7, w=0..2*Pi,
>                        coords=cylindrical);
>
```

Exercise: In third semester calculus you learn that the element of volume in cylindrical coordinates is given by $dV = r\,dr\,d\theta\,dz$. Draw a picture of an element of volume in cylindrical coordinates. (Hint: The element of volume has six faces. Draw three graphs, each with a pair of opposite faces, and then combine the three graphs together.)

```
>
```

Exercise: In third semester calculus you learn that the element of volume in spherical coordinates is given by $dV = \rho^2 \sin(\phi)\,d\rho\,d\theta\,d\phi$. Draw a picture of an element of volume in spherical coordinates.

```
>
```

Exercise: Here are eight different ways to graph a sphere. Explain how each one works.

```
> plots[implicitplot3d]( x^2+y^2+z^2=16, x=-4..4, y=-4..4,
  z=-4..4);
> plots[implicitplot3d]( rho=4, rho=0..4, theta=0..2*Pi,
  phi=0..Pi,
                         coords=spherical );
> plots[implicitplot3d]( r^2+z^2=16, r=0..4, theta=0..2*Pi,
  z=-4..4,
                         coords=cylindrical );
> plot3d( [4*sin(v)*cos(u),4*sin(v)*sin(u),4*cos(v)], u=0..2*Pi,
  v=0..Pi );
> plot3d( [x, sqrt(16-x^2)*cos(theta), sqrt(16-x^2)*sin(theta)],
          x=-4..4, theta=0..2*P i);
> plot3d( 4, theta=0..2*Pi, phi=0..Pi, coords=spherical );
> plot3d( sqrt(16-z^2), theta=0..2*Pi, z=-4..4,
  coords=cylindrical );
```

```
> plot3d( {sqrt(16-x^2-y^2), -sqrt(16-x^2-y^2)}, x=-4..4,
          y=-sqrt(16-x^2)..sqrt(16-x^2) );
>
```

When using **implicitplot** or **implicitplot3d**, the order of the ranges is very important. As some of the above exercises demonstrate, these commands use the order of the ranges to determine exactly which coordinate a variable represents. A variable named **r** need not represent radius in polar coordinates. A variable named **r** in an **implicitplot** equation will represent the radial coordinate in polar coordinates only if the first range given is for **r** (and of course the **coords=polar** option is used). On the other hand, there is nothing wrong with using the variable **x** to represent the radial coordinate in polar coordinates, we just have to list the range for **x** first (and use the **coords=polar** option).

```
>
```

Exercise: Explain why the following two graphs look the way they do.

```
> plots[implicitplot]( y=x^2-1, y=-1..3, x=-2..2 );
> plots[implicitplot]( x^2-1=y, x=-2..2, y=-1..3 );
```

Predict what the following graph will look like before drawing it.

```
> plots[implicitplot]( y^2-1=x, y=-1..3, x=-2..2 ):
>
```

We can use the ordering of ranges in **implicitplot** commands to find another way to draw nonstandard graphs of functions. Recall that if $f(s)$ is a real valued function of a single variable, then we can draw graphs of either $y = f(x)$ or $x = f(y)$ in Cartesian coordinates or we could draw graphs of either $r = f(\theta)$ or $\theta = f(r)$ in polar coordinates. By default, the **plot** command will draw only $y = f(x)$ in Cartesian coordinates and $r = f(\theta)$ in polar coordinates. Earlier we saw how to use the **plot** command with parametric equations to draw the graphs of $x = f(y)$ and $\theta = f(r)$. We can also draw these graphs using **implicitplot**. Here are the graphs of $x = \sin(y)$ and $\theta = \sin(r)$ drawn using **implicitplot**.

```
> plots[implicitplot]( x=sin(y), x=-1..1, y=0..2*Pi );
> plots[implicitplot]( theta=sin(r), r=0..4*Pi, theta=0..2*Pi,
                       coords=polar );
>
```

Exercise: Use **implicitplot** to draw the graph of $y = \sin(x)$ in Cartesian coordinates and the graph of $r = \sin(\theta)$ in polar coordinates.

```
>
```

Given a real valued function $f(u, v)$ of two real variables we can use the **implicitplot3d** command to draw any one of the six possible graphs of f in each of rectangular, cylindrical, and spherical coordinates.

```
```

```
>
```

Exercise: Use **implicitplot3d** to graph the function $z = \sin(x^2 + y^2)$ using cylindrical coordinates. Also graph this function using **plot3d** and both rectangular and cylindrical coordinates. Try to make the graphs as nearly equivalent as you can. Which graph turns out the "best", which the "worst"?

```
>
```

Exercise: For the function $f(u, v) = v \sin(u)$ use **implicitplot3d** with spherical coordinates to draw graphs of $\rho = f(\phi, \theta)$, $\theta = f(\phi, \rho)$, and $\phi = f(\theta, \rho)$. For each graph, find ranges for the variables that produce an interesting graph. Also, redraw each of these graphs using **plot3d** and parametric equations.

```
>
```

Earlier in this worksheet we drew contour diagrams for functions of two variables. These contour diagrams are closely related to graphs of equations. Suppose we have a function $f(x, y)$ of two variables. If we let c be any number, then we can make an equation of the form $f(x, y) = c$. The graph of an equation of this form is a level curve for the function f. A level curve for the function f is a curve in the plane such that the graph of f has constant elevation over this curve. If we use **implicitplot** to graph several level curves, then we get a contour diagram for f. Here is an example with the function $f(x, y) = x^2 - y^2$.

```
> f := (x,y) -> x^2-y^2;
> plots[implicitplot]( {f(x,y)=0, f(x,y)=2, f(x,y)=4,
>                                   f(x,y)=-2,f(x,y)=-4},
>                       x=-5..5, y=-5..5 );
```

Here is the equivalent **contourplot** command.

```
> plots[contourplot]( f(x,y), x=-5..5, y=-5..5,
  contours=[-4,-2,0,2,4] );
```

Notice that the equation $f(x, y) = c$ does not define a level set for the graph of f. We defined level sets as curves of constant elevation in three dimensional space and the equation $f(x, y) = c$ has its graph in two dimensional space. Here is a **plot3d** command that draws the level sets that are equivalent to the above level curves. Notice how you can rotate the next graph and see that these curves are really curves in three dimensional space and that they lie on the graph of f.

```
> plot3d( f(x,y), x=-5..5, y=-5..5,
>         style=contour, contours=[-4,-2,0,2,4],
>         orientation=[-90,0], axes=normal, view=[-5..5, -5..5,
  -5..5] );
>
```

```
>
```

5.10. Graphs of vector fields

Maple has two commands for drawing vector fields. The **fieldplot** command draws vector fields in the plane and **fieldplot3d** draws vector fields in space. Recall that a vector field in the plane is defined by a 2-dimensional vector valued function of two real variables. A vector field in space is defined by a 3-dimensional vector valued function of three real variables. Here is a simple example of a function that defines a vector field in the plane, $f(x, y) = (2x, 2y)$. If we let p_0 represent a point in the plane and we let (x_0, y_0) be the coordinates of p_0, then the value of $f(x_0, y_0)$ gives us the horizontal and vertical coordinates of a vector to be drawn at the point p_0. So at p_0 we would draw a vector with horizontal component $2x_0$ and vertical component $2y_0$. If we do this at every point p in the plane, then we will draw a vector field that always points away from the origin, and the length of every vector in the field is twice the distance of the base of the vector from the origin. Here is a graph of this vector field.

```
> plots[fieldplot]( [2*x, 2*y], x=-3..3, y=-3..3 );
>
```

Notice one thing right away about this graph. While the directions of the vectors in the graph are accurate, the lengths of the vectors are *not* accurate. For example, at the point (1,1) there is supposed to be a 45 degree vector with length $2\sqrt{2}$, which would put the tip of the vector past the point (3,3). But at (1,1) in the above graph there is a very short 45 degree vector. The vectors in the graph do get longer as they get further from the origin, but the lengths are nowhere near what they should be (why do you think that is?). A vector field drawn by **fieldplot** is not meant to be a literal representation of the true vector field. The graphs drawn by **fieldplot** are meant to give an impression of how a vector field looks. These graphs usually give us good qualitative (instead of quantitative) information about a vector field. In the above example, the graph shows us that every vector in the field points away from the origin and that the length of each vector is proportional to its distance from the origin, and this gives us a good idea of what the true vector field looks like.

```
>
```

Exercise: How would you expect the graphs of the vector fields $f(x, y) = (2x, 2y)$ and $g(x, y) = (3x, 3y)$ to differ? How would the graph of $h(x, y) = (-x, -y)$ compare with the graphs of f and g?

```
>
```

A vector field in the plane is defined by a single function but that function has two components, each of which is a real valued function of two variables. In the above **fieldplot** command we used two expressions in a list to describe the vector field, one expression for each component function. By making use of Maple functions, we can emphasize that a vector field is really defined by a single function. Here is a Maple function that defines the vector field used above

```
> f := (x,y) -> [2*x, 2*y];
```

Here is a **fieldplot** command that uses the Maple function **f**.

```
> plots[fieldplot]( f(x,y), x=-3..3, y=-3..3 );
```

Recall from the section on vector valued functions in the last worksheet that there is unfortunately no standard way to work with vector valued functions. So for example, the following obvious version of the last command does not work.

```
> plots[fieldplot]( f, -3..3, -3..3 );
```
And if we define **f** in the following common way (with parentheses instead of brackets)
```
> f := (x,y) -> (2*x, 2*y);
```
then we have to modify slightly the way we use **f** in **fieldplot**.
```
> plots[fieldplot]( [f(x,y)], x=-3..3, y=-3..3 );
```
The most common way to use **fieldplot** is to use expressions for each of the component functions of the vector field and for the most part that is the way we will work with the command (just as we did with parametric curves and parametric surfaces, both of which also use vector valued functions).
```
>
```

Here are some examples of the kinds of vector fields that come up in a course on vector calculus or differential equations.
```
> plots[fieldplot]( [-y, x], x=-5..5, y=-5..5 );
> plots[fieldplot]( [ln(1+y^2), ln(1+x^2)], x=-5..5, y=-5..5 );
> plots[fieldplot]( [x/2, -y/3], x=-2..2, y=-2..2 );
```
Here are a few 3-dimensional vector fields. It is not easy to get much information out of these kinds of graphs.
```
> plots[fieldplot3d]( [y, z, x], x=-2..2, y=-2..2, z=-2..2 );
> plots[fieldplot3d]( [y/z, -x/z, z/4], x=-2..2, y=-2..2, z=1..3
  );
> plots[fieldplot3d]( [-x, -y, -z], x=-2..2, y=-2..2, z=-2..2 );
>
```

There is an important special class of vector fields that comes up very often in mathematics, gradient vector fields. This kind of vector field is used often enough that Maple has two special commands for drawing them, **gradplot** for 2-dimensional gradient fields and **gradplot3d** for 3-dimensional gradient fields.

Recall from third semester calculus that if we have a real valued function of two real variables, then its derivative at a point is a vector whose two components are the two partial derivatives of the function at the point. If we compute the derivative of the function at every point in its domain, then we get a vector field of gradient vectors. This vector field is called a **gradient field** and the original function is called gradient field's **potential function**. Let us look at an example. Consider the function $f(x, y) = \sin(x + y)$. Here are two ways to draw its gradient field, using **fieldplot** and using **gradplot**. First, let us define the function to Maple.
```
> f := (x,y) -> sin(x)+sin(y);
```
Here is how we can compute its two partial derivative functions.
```
> D[1](f); D[2](f);
```
Here is how we can use **fieldplot** to draw the gradient vector field for **f**. We give **fieldplot** the two partial derivatives of **f** as the components of the vector field.
```
> plots[fieldplot]( [ D[1](f), D[2](f) ], -6..6, -6..6 );
```

Here is how we use **gradplot** to draw the same gradient vector field for **f**. We only need to give **gradplot** the potential function.

```
> plots[gradplot]( f(x,y), x=-6..6, y=-6..6 );
>
```

Recall from calculus that if p is any point in the domain of the potential function $f(x, y)$, then the gradient vector at p is perpendicular to the level curve passing through p. Let us demonstrate this by drawing both the gradient field and the contour diagram for **f** in the same graph. (The parameter **b** is there to make it easier to change the domain of the graph.)

```
> b := 4:
> plots[contourplot]( f(x,y), x=-b..b, y=-b..b ):
> plots[gradplot](    f(x,y), x=-b..b, y=-b..b ):
> plots[display]( %, %% );
```

If you look closely at the above graph, you can see that all of the gradient vectors are perpendicular to the level curves. (This is a bit easier to see if you zoom in on the graph a bit.) A bit more can be said about the directions of the gradient vectors. Recall that the color coding on level curves goes from red for low values to yellow for high values. The gradient vectors are pointing from red curves toward yellow curves. This shows, as you learned in calculus, that the gradient vectors point in the direction of steepest increase in the potential function. To help you see that the gradient vectors are pointing "uphill", compare the above graph with the following three dimensional graph of **f** and its level sets.

```
> b := 4:
> plot3d( f(x,y), x=-b..b, y=-b..b, style=contour,
  orientation=[-90,0] );
>
```

Exercise: Draw a combined graph of the level curves and gradient field for the potential function $f(x, y) = \frac{3y}{x^2 + y^2 + 1}$.

```
>
```

So far in this section, we have always considered a function of the form $(u, v) = f(x, y)$ as representing a vector field in the plane, that is u and v are the horizontal and vertical components of a vector that we draw at the point with coordinates (x, y). But there are other ways of interpreting this kind of function. Let us look briefly at one of these other interpretations. We can use a 2-dimensional vector valued function of two variables as a **change of variables**. Here is an example. Consider the function $(u, v) = f(r, \theta)$ defined by $f(r, \theta) = (r\cos(\theta), r\sin(\theta))$. If we let p represent a point in the plane, and suppose that p has polar coordinates (r, θ), then $(u, v) = (r\cos(\theta), r\sin(\theta))$ will be the Cartesian coordinates for p. In other words, given a point in the plane, the function f takes as input the polar coordinates of the point and returns the Cartesian coordinates for the same point. The function f changes the polar coordinates of a point into Cartesian coordinates, and so we call it a change of variables. Of course, the function f can also be given a

vector field interpretation. Let us demonstrate using both of these interpretations for f. Let us define f to Maple.

```
> f := (r,theta) -> [r*cos(theta), r*sin(theta)];
```

Here is a list of four points in the plane given in polar coordinates. These point are at the four corners of a square.

```
> points := [ [sqrt(2),   Pi/4], [sqrt(2), 3*Pi/4],
              [sqrt(2), 5*Pi/4], [sqrt(2), 7*Pi/4] ];
```

Let us graph these points using polar coordinates.

```
> plot( points, style=point, coords=polar );
```

Now let us apply the change of variables function **f** to these points and get a list of the Cartesian coordinates for the points. In the following command, **p** represents an ordered pair from the list **points**, **p[1]** is the first number in **p** (i.e., the radial coordinate) and **p[2]** is the second number in **p** (i.e., the angular coordinate).

```
> seq( f(p[1],p[2]), p=points );
```

Now graph the points using Cartesian coordinates.

```
> plot( [%], style=point );
```

We just used **f** to change the coordinates of four points in the plane from polar to Cartesian. Now let us interpret **f** as a vector field and graph the vector field.

```
> plots[fieldplot]( f(x,y), x=-6..6, y=-6..6 );
```

So we have given the same function **f** two very different interpretations, as a change of variables and as a vector field. Neither interpretation is more correct than the other. Some 2-dimensional vector valued functions of two variables are more useful as a vector field and some are more useful as a change of variables.

```
>
```

Exercise: The function $T(\rho, \theta, \phi) = (\rho \sin(\phi) \cos(\theta), \rho \sin(\phi) \sin(\theta), \rho \cos(\phi))$ can be interpreted as a change of variables or as a vector field. Create a short list of spherical coordinates for some points in space, plot the list using spherical coordinates, then use T to change the coordinates to rectangular coordinates and plot the points using rectangular coordinates, and then graph T as a vector field. (Note: To plot points in space, you need to use the **pointplot3d** command from the **plots** package. The **plot3d** command does not plot points.)

```
>
```

```
>
```

5.11. Online help for graphing and visualization

Maple has extensive graphing abilities. Besides the two main graphing commands, **plot** and **plot3d**, and the main package of graphing commands, **plots**, Maple has a lot of other graphing facilities scattered throughout a number of packages. There is a lot of online documentation and examples for these facilities. Below we try to outline this documentation. First we mention the documentation for the **plot** and **plot3d** commands and their most important options, and then we outline the documentation for the most important commands within the **plots** package. (Notice, as

you go along, that a help page like **?plot,coords** is about an *option* to the **plot** command, and a help page like **?plots,coordplot** is about a *command* in the **plots** package.) Finally, at the end of this section we try to outline most of the documentation for Maple's other graphing commands and packages.

When you click on any two-dimensional or three-dimensional graph, the menus available at the top of the Maple window and the context bar just below the menus change appropriately. Here are two help pages that describe the items in the graphics menus.

```
> ?worksheet,reference,plot2dmenu
> ?worksheet,reference,plot3dmenu
```

And here are two help pages that describe the items in the graphics context bars.

```
> ?worksheet,reference,context2dplot
> ?worksheet,reference,context3dplot
```

Here are two more help pages that give information about modifying graphs using menus.

```
> ?style2
> ?style3
```

At this point we should mention that on a few platforms Maple has defined a special kind of plot, a "smart plot," that is supposed to make it easier to draw basic graphs. The following help page gives an overview of the two kinds of Maple plots, "standard plots" and "smart plots".

```
> ?worksheet,plotinterface,plottypes
```

There are several ways to create smart plots. They can be created by right clicking on the output from some Maple commands.

```
> ?worksheet,plotinterface,interactive
> ?worksheet,plotinterface,contextmenu
```

And smart plots can be created directly by using several different commands.

```
> ?plots,interactive
> ?smartplot
> ?smartplot3d
```

An interactive graph drawn by these commands can be manipulated by right clicking on the graph to bring up a context menu. These context menus have more items in them than the context menus for "standard plots".

Maple's most basic graphing command is of course **plot**.

```
> ?plot
```

The **plot** help page has almost no information about the options to **plot**. These are all described in the following important help page.

```
> ?plot,options
```

Many of the special features of the **plot** command, like parametric graphs, polar graphs, using infinity in a range, etc., have their own help pages.

```
> ?plot,multiple
> ?plot,parametric
> ?plot,function
```

```
> ?plot,ranges
> ?plot,infinity
> ?plot,color
> ?plot,style
> ?plot,polar
> ?plot,coords
```

Most of the options to the **plot** (and **plot3d**) command translate directly into pieces of a PLOT data structure. In a later worksheet we will say more about PLOT data structures. The following command brings up a description of the PLOT data structure and all of its pieces. Sometimes, looking up the PLOT data structure analogue of a **plot** option will provide some clue about the option that is not in the option's documentation.

```
> ?plot,structure
```

Another description of various plot options, or "plot attributes," is given in the following help page and its hyperlinks.

```
> ?worksheet,plotinterface,plotattributes
```

We mentioned the **discont=true** option of the **plot** command. This option does not have its own help page. But the **discont=true** option to **plot** makes use of a Maple function called **discont**. The following help page is about the **discont** function and therefore it also sheds some light on the **discont** option of **plot**.

```
> ?discont
```

The **fdiscont** function, which is also used by **plot** with the **discont=true** option, does the same thing as **discont** but it works numerically instead of symbolically.

```
> ?fdiscont
```

Maple's basic three dimensional graphing command is **plot3d**.

```
> ?plot3d
```

The **plot3d** help page says very little about the options to the **plot3d** command. All of these options are explained in the next help page.

```
> ?plot3d,options
```

The **plot3d** command has two other help pages about some of its special features.

```
> ?plot3d,coords
> ?plot3d,colorfunc
```

The **plot3d** command has the **style=contour** option for drawing the level curves of a surface. Two closely related commands from the **plots** package are **contourplot** and **contourplot3d**. The **contourplot** command draws all of the level curves of a surface as level sets in the plane. The **contourplot3d** command draws the level curves as curves in space. At first glance the **contourplot3d** command may seem to be equivalent to the **plot3d** command with the **style=contour** option but in fact they are a bit different. In particular, the **filled** option works differently in **plot3d** and **contourplot3d**, and **contourplot3d** has

the **coloring** option. The following help page describes both **contourplot** and **contourplot3d**.

```
> ?plots,contourplot
```

If you are drawing a lot of graphs and they are all using the exact same options, it might be convenient to redefine the default **plot** and **plot3d** options using the **setoptions** and **setoption3d** commands from the **plots** package.

```
> ?plots,setoptions
> ?plots,setoptions3d
```

Maple can draw graphs in 15 two-dimensional coordinate systems and 31 three-dimensional coordinate systems. The next help page lists all of these coordinate systems, and, additionally, gives the formulas for the coordinate transformation of each coordinate system to Cartesian coordinates.

```
> ?coords
```

The next two help pages briefly summarize the two and three dimensional coordinate systems respectively.

```
> ?plot,coords
> ?plot3d,coords
```

The next two help pages describe the commands for drawing pictures of the two and three dimensional coordinate systems. For the two-dimensional coordinate systems, the **coordplot** command draws a picture of "graph paper" for each coordinate system. For the three-dimensional coordinate systems the **coordplot3d** command draws a surface of constant value for each of the three coordinate variables.

```
> ?plots,coordplot
> ?plots,coordplot3d
```

The command for defining your own coordinate systems is **addcoords**.

```
> ?addcoords
```

It is worth mentioning that the **plots** package contains three functions for graphing in non Cartesian coordinate systems that seem to be redundant with options for the **plot** and **plot3d** commands. The **polarplot** command seems to be equivalent to the **plot** command with the **coords=polar** option, the **cylinderplot** command seems to be equivalent to the **plot3d** command with the **coords=cylinder** option, and the **sphereplot** command seems to be equivalent to the **plot3d** command with the **coords=spherical** option.

```
> ?plots,polarplot
> ?plots,cylinderplot
> ?plots,sphereplot
```

Closely related to the idea of using different coordinate systems in the plane or in space is the idea of using a different coordinate system on the real line when graphing a real valued function of one variable. As the following three help pages describe, Maple can draw graphs of real valued functions of a single variable with a logarithmic scale on either or both of the axes.

```
> ?plots,logplot
```

```
> ?plots,semilogplot
> ?plots,loglogplot
```

The **plot3d** command only draws graphs of surfaces, that is, graphs of real valued functions of two variables and parametric surfaces. To plot points in three dimensions, Maple needs a special command, **pointplot3d**, from the **plots** package.

```
> ?plots,pointplot3d
```

To draw curves in three dimensions Maple needs a special command, **spacecurve**, from the **plots** package

```
> ?plots,spacecurve
```

The **tubeplot** command lets us convert a one dimensional curve in three dimensional space into a two dimensional "tube".

```
> ?plots,tubeplot
```

There are two commands for graphing equations, one that draws two dimensional graphs of equations in two variables, and one that draws three dimensional graphs of equations in three variables.

```
> ?plots,implicitplot
> ?plots,implicitplot3d
```

There are two commands for graphing vector fields, one for two dimensional vector fields in the plane and one for three dimensional vector fields in space.

```
> ?plots,fieldplot
> ?plots,fieldplot3d
```

In addition, there are two commands for the special case of drawing gradient vector fields.

```
> ?plots,gradplot
> ?plots,gradplot3d
```

One very important command from the **plots** package is the **display** command, which can be used to combine several (usually simple) graphs into a more complicated graph.

```
> ?plots,display
```

One way that **display** can combine several graphs together is as the frames of an animation. This is done by using the **insequence=true** option to **display**. There is no separate help page for the **insequence** option. The previous help page includes a description of this option and several examples of its use. Another way to create animations is by using the **animate** and **animate3d** commands. These commands provide an easy way to make simple animations, but they are not as versatile as the **insequence=true** option to **display**.

```
> ?animate
> ?animate3d
```

There is a special animate command specifically for animating curves in the plane. In particular, this command makes nice animations of parametric curves.

```
> ?animatecurve
```

Maple animations can be used to create animated GIF files for use in web pages on the Internet. A brief explanation of this, along with an explanation of some other graphics formats that Maple can produce, is in the next help page.

[> **?plot,device**

The plot devices described in the last help page are used as options in either the **plotsetup** or **interface** commands.

[> **?plotsetup**

[> **?interface**

An interesting command from the **plots** package is **matrixplot**, which lets you draw a three dimensional visualization of the contents of a matrix.

[> **?plots,matrixplot**

In the New User's Tour there is a worksheet containing examples of using some basic graphics commands.

[> **?newuser,topic05**

Besides the **plot** and **plot3d** commands and the commands mentioned above from the **plots** package, Maple has many other commands for drawing graphs. In the rest of these paragraphs we show where to get additional information about some of these commands.

There are many graphing commands in the **plots** package that we have not yet mentioned. The next page is a summary of the entire **plots** package and it contains hyperlinks to the help pages of all the commands in the package.

[> **?plots**

The **plottools** package has a number of special functions for drawing "graphical objects", like an arrow. The next page is an overview of this package and it contains hyperlinks to the help pages of the all the commands in the package.

[> **?plottools**

Maple also has a **geometry** package that can be used to create illustrations of ideas and theorems from two dimensional Euclidean geometry. Here is a help page giving an overview of this package.

[> **?geometry**

Within this package you can work with the following kinds of geometric objects.

[> **?geometry,objects**

You can apply the following types of transformations to the geometric objects.

[> **?geometry,transformation**

The **draw** command in the **geometry** package is used to actually draw the geometric objects that you define using the package's commands.

[> **?geometry,draw**

Here is a help page that contains several examples of using the **geometry** package. You can cut

each example out of the help page and paste it into a worksheet and then execute the example.

```
> ?geometry,examples
```

And here is a worksheet that has more examples of the **geometry** package. You can execute these examples directly in this worksheet.

```
> ?examples,geometry
```

For doing Euclidean geometry in three dimensions Maple has the **geom3d** package. Here is an overview of this package.

```
> ?geom3d
```

Within this package you can work with the following kinds of geometric objects.

```
> ?geom3d,objects
```

You can apply the following types of transformations to the geometric objects.

```
> ?geom3d,transformation
> ?geom3d,transform
```

The **draw** command in the **geom3d** package is used to actually draw the geometric objects that you define using the package's commands.

```
> ?geom3d,draw
```

Here is a worksheet that has some examples of the transformations available in the **geom3d** package. You can execute these examples directly in this worksheet.

```
> ?examples,transform
```

Maple has many commands for working with polyhedra. These commands are contained in three packages, **plots**, **plottools**, and **geom3d**. I do not really understand the division of labor involved here. I'll try to give some pointers to the documentation. In the **plots** package there is the **polyhedraplot** command.

```
> ?plots,polyhedraplot
```

There is a command that will list the polyhedra supported by the **polyhedraplot** command.

```
> ?polyhedra_supported
```

Here is the list of the polyhedra supported by **polyhedraplot**. It is interesting to see the names of so many kinds of polyhedra. (Can you say parabidiminishedrhombicosidodecahedron?)

```
> polyhedra_supported();
```

In the **plottools** package there are five commands that act as an interface to the **polyhedraplot** command.

```
> ?plottools,dodecahedron
> ?plottools,hexahedron
> ?plottools,icosahedron
> ?plottools,octahedron
> ?plottools,tetrahedron
```

The **plottools** package has a few commands for modifying polyhedra, for example the **stellate** command.

```
> ?plottools,stellate
```

The **geom3d** package has a number of commands that can be used to define polyhedra. The

polyhedra defined with these commands can be drawn using the **draw** command from the same package. The commands for creating polyhedra are organized into three groups.

```
> ?geom3d,RegularPolyhedron
> ?geom3d,QuasiRegularPolyhedron
> ?geom3d,Archimedean
```

The **geom3d** package has a command for creating the dual of a given polyhedron.

```
> ?geom3d,duality
```

Like the **plottools** package, the **geom3d** package has a command to stellate a polyhedron.

```
> ?geom3d,stellate
```

In addition, the **geom3d** package has a command for faceting a polyhedron.

```
> ?geom3d,facet
```

There are six worksheets that give examples of how to use the **geom3d** package to work with polyhedra. You can execute the examples directly in these worksheets. Some of the examples in these worksheets create very striking polyhedra.

```
> ?examples,regular
> ?examples,archi
> ?examples,dual
> ?examples,stellate
> ?examples,facet
> ?examples,transform
```

Here are several other packages with some specialized plotting and visualization commands in them.

```
> ?student
> ?Student,Calculus1
> ?Student,Calculus1,VisualizationOverview
> ?DEtools
> ?PDEtools
> ?stats,statplots
```

Here are two worksheets that contain examples of using the **DEtools** package to graph solutions of differential equations. You can execute the examples directly in these worksheets.

```
> ?examples,deplot
> ?examples,deplot3d
```

Here is a worksheet with some examples from the **statplots** (sub) package.

```
> ?examples,statplots
```

```
>
```