Maple for Math Majors

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

6. How Maple draws graphs

6.1. Introduction

By now you should have a good sense of how to create most of the kinds of graphs that you need in a math or science course. But one of the goals of these worksheets is to give you a sense of how Maple does what it does, a sense of what is going on inside Maple as it produces its results. So in this worksheet we look at how Maple's basic graphing commands work. We examine some, but not all, of the details of what these commands go through to create a graph. In the sections below we look at how the `plot` command draws graphs of real and vector valued functions of a single variable, how the `implicitplot` command draws graphs of equations in two variables, how the `plot3d` command draws surfaces defined by real valued functions of two variables, and how the animation commands create movies. The material in these sections is important for several reasons. Besides giving you an idea of how Maple works, this material helps you understand some of the anomalies and problems that can come up when using Maple's graphing commands. This material is also a good way to practice using and thinking about Maple's graphing commands. And this material should also help you to better understand the mathematical nature of functions, equations, and graphs.

What we go over is these sections is not the whole story of how Maple draws graphs. We will go into some more of the details after we have discused the basics of Maple's data structures in a later worksheet.

```
>
```

6.2. The `plot` command

What does Maple do to draw a graph? The answer is surprisingly simple. Maple does what just about anyone would do if asked to draw a graph. Maple plots a bunch of points and then connects them together with straight lines. To help see that this is the case, there is an option to the `plot` command that tells Maple to turn off the straight lines that connect the plotted points. Here is an example.

```
> plot( sin(x), x=-Pi/2..Pi/2, style=point, symbol=point,
  axes=none );
```

The `style=point` option turned off the lines. (And `style=line` turns them back on. Try it.) The `symbol=point` option tells Maple to actually plot points and not some other symbol. The `axes=none` option turns off the drawing of the two axes, so that it is easier to see the points. But the points are still pretty faint and hard to see, so let us use a different symbol to make the position of the points more clear.

```
> plot( sin(x), x=-Pi/2..Pi/2, style=point, symbol=circle,
  axes=none );
```

Here are a few more examples of Maple plots without the line segments that should connect the dots.

```
> plot( x^2 , x=-2..2 , style=point, symbol=circle );
> plot( 1/x, x=-10..10, -10..10, style=point, symbol=circle,
  axes=none );
```

Note: You can put the axes back in by clicking on the graph and then clicking on one of the axes buttons in the context bar.

```
> plot( sin(1/x), x=.05..1, style=point, symbol=circle, axes=none
  );
```

Play with these examples a bit. Use the `style` option to turn on and off the line segments. (You can change the `style` option four ways. By modifying the Maple command. By clicking on the graph and using the Style menu or the style buttons on the context bar. And by right clicking on the graph and choosing Style from the popup context menu.) Change the ranges of the graphs to see how that affects the positions of the plotted points. Try other functions.

```
>
```

We can even do this with 3-dimensional graphs. Be sure to rotate each of the following examples. Try changing the `style` option. In the 3-D case there are more `style` options to choose from. Try `line`, `contour`, and `patch` (if you try `patch`, then you will need to get rid of the `color` option).

```
> plot3d( x^2+y^2, x=-4..4, y=-4..4, style=point, color=black );
> plot3d( x^2-y^2, x=-4..4, y=-4..4, style=point, color=black );
> plot3d( sin(x*y), x=-Pi..Pi, y=-Pi..Pi, style=point,
  color=black );
```

In another section we will look at how Maple draws surfaces in more detail.

```
>
```

Let us look at an example of why it is useful to understand how Maple draws graphs. Consider the following graph of the function $1/(1-x)$.

```
> plot( 1/(x-1), x=-4..6, -20..20, color=red );
```

Notice the vertical red line. That line is not supposed to be part of the graph. Why is it there? Let us look at the points that Maple is plotting.

```
> plot( 1/(x-1), x=-5..5, -20..20, style=point, color=red );
```

This should answer our question. Maple plots points and then connects all successive points together with straight lines. This includes the two extreme points on either side of the vertical asymptote at $x = 1$. They get connected together in an almost vertical straight line that should *not* be there. Maple has a way to avoid this problem. The plot option `discont=true` tells Maple to avoid drawing these extra lines at vertical asymptotes.

```
> plot( 1/(x-1), x=-5..5, -20..20, discont=true, color=red );
```
Now the vertical red line is gone. Maple does not have this option on by default because it is an "expensive" option. It forces Maple do quite a bit of extra calculating to find the discontinuities of the function. So we must turn this option on ourselves whenever we need it.
```
>
```

Now let us look further into how Maple draws 2-dimensional graphs. It seems that Maple has a pretty simplistic way of drawing these graphs. But it turns out that Maple is not so simple minded. Here is what Maple is concerned about. The more dots that are used, the better looking the graph will be, but the more dots that are used the more work (calculations) Maple will need to do. How many dots might a graph need to look good? Maple tries to use only 49 points for a 2-dimensional graph and Maple tries to be very clever about how it uses those points. To decide where best to put those dots Maple uses what is called an **adaptive plotting algorithm**. This means that Maple does not space out its points uniformly along the horizontal axis. Maple tries to adapt the horizontal spacing of the points so that the points are closer together in some places and farther apart in other places. Where would Maple need fewer dots spaced farther apart? Since the dots are connected by straight lines, wherever the graph of a function is nearly straight Maple does not need to use many points to get a good looking graph (think of the extreme of graphing a straight line). Where would Maple need more points spaced close together? Wherever the graph is changing directions fairly quickly. Let us go back to our first example.
```
> plot( sin(x), x=-Pi/2..Pi/2, style=point, symbol=circle,
  axes=none );
```
If you look closely you will notice that the points in the middle of the graph, where the sine function is almost linear, are more spaced out than the points near the ends of the graph where the sine function has turning points. But the difference is quite subtle in this case. Let us try to find a more extreme example. We want a function that is very linear-like over a large interval and then suddenly changes to a sharply turning graph. To try to get such an example, let us see if we can exagerate these properties of the sine function. What gives the sine function its overall shape near the origin? The first few terms of its Taylor series about the origin.
```
> taylor( sin(x), x=0 );
```
Let us see what happens if we use just the first two terms of this series.
```
> plot( x-(1/6)*x^3, x=-Pi/2..Pi/2, style=point, symbol=circle,
  axes=none );
```
Almost exactly the same graph. How can we exagerate the linearity vs. the nonlinearity in this example? After some experimenting with the power and the coefficient of the second term, here is a pretty good example to look at.
```
> plot( x-(1/1000)*x^21, x=-1.35..1.35, -2..2, style=point,
  symbol=circle, axes=none );
```
Notice the long stretch of near linearity and then the very sharp turn where points are bunched up. So we can see how Maple is adaptively plotting points. Maple allows us to use the `plot` option `adaptive=false` to turn off the adaptive point plotting. In the next example, the points are spaced uniformly along the horizontal axis. Compare this graph to the previous one.
```
> plot( x-(1/1000)*x^21, x=-1.35..1.35, -2..2, adaptive=false,
  style=point, symbol=circle, axes=none );
>
```

Let us look at another example of adaptive vs. non adaptive point plotting. We can use the `plot` option `numpoints` to control how many points we want plotted. In the case of non adaptive plotting, `numpoints` determines exactly how many points we want Maple to plot. In the case of adaptive plotting, `numpoints` is only a lower bound on the number of points plotted, but in most cases the exact number plotted is near to `numpoints`. The next example is a slight modification of the previous example.

```
> plot( x+.0001*x^21, x=0..1.6, 0..4, numpoints=20, style=point,
  symbol=circle );
```

Here we can see equally spaced points along the linear portion of the graph and bunched up points near the truning part of the graph. We made `numpoints` 20 but there are 29 points in the graph (this example is convenient because we can distinctly see all the point and count them). The next example sets `adaptive=false` and sets `numpoints=29` so the next graph has exactly the same number of points as the previous graph.

```
> plot( x+.0001*x^21, x=0..1.6, 0..4, adaptive=false,
  numpoints=29, style=point, symbol=circle );
```

To really see the difference between the last two graphs, let us combine them together into one graph.

```
> p1 := plot( x+.0001*x^21, x=0..1.6, 0..4, numpoints=20,
  style=point, symbol=circle, color=black ):
> p2 := plot( x+.0001*x^21, x=0..1.6, 0..4, adaptive=false,
  numpoints=29, style=point, symbol=circle, color=blue ):
> plots[display]( p1, p2 );
```

The black circles are the adaptively plotted points and the blue circles are the uniformly spaced points. Here is another way to compare these two graphs. The next graph once again combines the adaptive and non adaptive graphs, but this time we shift one of the graphs up a bit to make it easier to see how points line up between the two graphs.

```
> p1 := plot( x+.0001*x^21+.1, x=0..1.6, 0..4, numpoints=20,
  style=point, symbol=circle, color=black ):
> p2 := plot( x+.0001*x^21, x=0..1.6, 0..4, adaptive=false,
  numpoints=29, style=point, symbol=circle, color=blue ):
> plots[display]( p1, p2 );
```

Now we can really see that the non adaptive graph (blue circles) has too many points in the linear portion of the graph and not enough points in the nonlinear portion of the graph.

```
>
```

Here is another example of adaptive vs. non adaptive plots that is interesting to play with.

```
> plot( x-.0001*x^21, x=0..1.6, 0..4, numpoints=20, style=point,
  symbol=circle );
> plot( x-.0001*x^21, x=0..1.6, 0..4, adaptive=false,
  numpoints=37, style=point, symbol=circle );
```
In this last example, try incrementing the value of `numpoints` in the adaptive plot to see how the extra points get used.
```
>
```

Here is one last example. When we graph the sine function over a whole period the adaptive plotting becomes very evident at the critical points. Try several periods of sine.
```
> plot( sin(x), x=-Pi..Pi, style=point, symbol=circle, axes=none
  );
> plot( sin(x), x=-Pi..Pi, adaptive=false, style=point,
  symbol=circle, axes=none );
>
```

We have been using the `numpoints` option to help us see the affects of adaptive plotting. But `numpoints` is usually used to help Maple draw better graphs. With some unusual functions, even Maple's adpative plotting algorithm will fail to plot enough points and the graph of the function gets distorted. In these cases, setting `numpoints` to some high number fixes the graph. Here are some examples.

Here is a typical example of a need for setting a value for `numpoints`.
```
> plot( x/(1-cos(5*x)), x=-5..5, -5..5 );
```
Notice that the critical points of the graph have a lot of sharp corners near them. The use of `numpoints` fixes this.
```
> plot( x/(1-cos(5*x)), x=-5..5, -5..5, numpoints=200 );
```
Notice that the function `x/(1-cos(5*x))` has a lot of vertical asymptotes. It turns out that we could have fixed our original graph of this function by using the `discont` option instead of `numpoints`.
```
> plot( x/(1-cos(5*x)), x=-5..5, -5..5, discont=true, color=red
  );
```
Go back and convert each of the last three graphs to `style=point` and see where the adaptive plotting algorithm was putting its ponts.
```
>
```

Here is a typical 3-dimensional example of a need for `numpoints`. Let `f` represent a parametric curve in space.
```
> f := [cos(t), sin(t),
  cos(t)^3*sin(4*sin(t))+sin(t)^2*cos(3*cos(t))];
> plots[spacecurve]( f, t=0..2*Pi );
```
Look at the graph carefully. You should notice several sharp corners near the turning points. We can improve this graph by using numpoints.
```
> plots[spacecurve]( f, t=0..2*Pi, numpoints=100 );
```
Now the graph is nice and smooth. Experiment with this last example a bit. Try incrementing

`numpoints` by 10's starting at 50 and find out when the graph no longer has any sharp corners at the turning points.

```
> 
```

Here is an even more dramatic example of a need for `numpoints`. Using the same parametric curve `f`, let us try increasing the range of the parameter `t`. This should not have any affect on the graph, since the function `f` is periodic, but watch what happens.

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

What do you think happened? What caused the curve in the graph to get "doubled up"? Try fixing this last graph using `numpoints`. The next graph is even stranger.

```
> plots[spacecurve]( f, t=0..7*Pi );
```

What do you think happened in this graph? What caused the graph to loose so much if its shape? How does this example differ from the last one? Try an even longer range for the parameter `t`.

```
> 
```

The number of points that Maple needs to use for graphing equations can be surprising. The next graph is a bit rough, so it needs to have a higher value for `numpoints`. Start with `numpoints=200`. How high do you need to go?

```
> plots[implicitplot]( x^3+y^3=5*x*y+1-5/4, x=-3..3, y=-3..3 );
> 
```

Here is an unusual example of a need for `numpoints` (this is from *Introduction to Maple*, 3rd Ed., by A. Heck, page 439).

```
> plot( (1/10)*(x-25)^2+cos(2*Pi*x), x=0..49 );
```

The graph looks just like a parabola. But it should not. Use a high value of numpoints.

```
> plot( (1/10)*(x-25)^2+cos(2*Pi*x), x=0..49, numpoints=200 );
```

This is the correct graph. The cosine term in the function adds a waviness to the parabolic term. In the original graph, all of the sample points being plotted were on the parabola part of the graph. The cosine term was nearly zero at every one of them!

```
> 
```

Here is a very extreme example of a use of `numpoints`. The following graph tries to plot 95 periods of the sine function. This is far more periods than can be realistically drawn on the computer screen but the graph is much more bizarre than it should be.

```
> plot( sin(x), x=-90*Pi..100*Pi );
```

The next example tries to fix the previous graph by setting `numpoints` very high. The graph looks much more regular now, though it still has an unusual appearance. Try changing the number of periods of sine in both the previous graph and this next graph and try changing the value of `numpoints` in the next example. The graphs you get are not very meaningful as far as the sine function is concerned, but they can be interesting to look at and to try and figure out where the patterns come from. (To get another sense of how strange this example is, click on the following graph and then click on each of the axes buttons on the context bar and see what happens.)

```
> plot( sin(x), x=-90*Pi..100*Pi, numpoints=1000 );
>
```

Exercise: The following command turns off adaptive plotting and draws sin(x) with just a few points so that you can see the straight lines that make up the graph.

```
> plot( sin(x), x=0..Pi, adaptive=false, numpoints=5 );
```

Increase the value of `numpoints` in small increments until you can no longer see any change in the graph. Then turn the line drawing off using the `style=point` option, so that you can see just how far apart the individual points are. Try the same experiment agian, starting with `numpoints` very low, but with a larger range for the variable `x`.

```
>
```

Exercise: Here is a tricky Maple puzzle. The real valued function $\sqrt{1-x}$ is not defined when x is greater than one. Maple can graph this function from 0 to 1 with no problem.

```
> plot( sqrt(1-x), x=0..1 );
```

Suppose we ask Maple to graph the function a little bit past 1, just to see what happens.

```
> plot( sqrt(1-x), x=0..1.01 );
```

Notice that we lost the last little bit of the graph *before* 1 (and the vertical asymptote at $x = 1$ is not as evident). Why do you think that happened? Notice that a high value of `numpoints` "fixes" this problem, though the original graph, which did not even need `numpoints`, is still better.

```
> plot( sqrt(1-x), x=0..1.01, numpoints=200 );
>
```

Exercise: Consider the following graph.

```
> plot( sin(100*x), x=-10*Pi .. 10*Pi, numpoints=500,
  adaptive=false );
```

Now increment `numpoints` from 500 to 510 in steps of one. Can you give any kind of an explanation for these graphs?

```
>

>
```

6.3. The `implicitplot` command

Of all the basic graphing commands in Maple, the `implicitplot` command has the most difficult job to do. In fact it is not uncommon for the `implicitplot` command to draw an incorrect graph, or even no graph, for an equation. Here is a fairly simple example. Let us try to graph the equation

$$(x^2+y^2)\,\mathbf{e}^{(1-(x^2+y^2))} = 1$$

which has the following simpler equation in polar coordinates

$$r^2\,\mathbf{e}^{(1-r^2)} = 1.$$

Before graphing this equation, let us load the **plots** package, since **implicitplot** is in this package.

```
> with(plots);
```

Here is the **implicitplot** command that tries to graph our equation.

```
> implicitplot( (x^2+y^2)*exp(1-(x^2+y^2))=1, x=-1..1, y=-1..1 );
```

The command produced an empty graph. But what does the graph of this equation look like? Let us solve the equation using its polar coordinate form. If $r = 0$, then the equation is not true, so we can assume the $r \neq 0$. If $r \neq 0$, we can simplify the equation to $\mathbf{e}^{(1-r^2)} = 1$, which is only solved by $1 - r^2 = 0$, which is solved by the curve given in polar coordinates by $r = 1$. In other words, the graph of the equation we gave to the **implicitplot** command is a circle of radius one centered at the origin. That is not a very complicated graph, so it is a bit surprising that **implicitplot** could not graph it. To make things even more surprising, let us show that Maple can solve the equation symbolically.

```
> solve( (x^2+y^2)*exp(1-(x^2+y^2))=1, {x,y} );
```

Maple found two symbolic solutions, the upper and lower halves of the circle of radius one centered at the origin. So this equation is not by any means very complicated, and its graph is certainly not very complicated, so why does the **implicitplot** command have trouble with it? To answer this question we need to look into how the **implicitplot** command works. After we have explained how **implicitplot** works, we will return to this example and a few other examples that give **implicitplot** problems.

```
>
```

Exercise: Draw a graph for the expression **(x^2+y^2)*exp(1-(x^2+y^2))** as a function of two variables. Where is the solution to the equation in the graph of this expression? Using the graph of the expression, can you see anything unusual about the solution to the equation that might be causing **implicitplot** problems?

```
>
```

Recall that we have defined an equation as an equal sign with an expression on either side of it. The **implicitplot** command can only graph equations in two variables, so as far as the **implicitplot** command is concerned, an equation is always of the form $\mathrm{f}(x, y) = \mathrm{g}(x, y)$, where we are using the notation $\mathrm{f}(x, y)$ and $\mathrm{g}(x, y)$ to represent expressions (*not* functions) in x and y (any two variables can be used; we settle on x and y just for convimience). The very first thing that the **implicitplot** command does is rewrite the equation $\mathrm{f}(x, y) = \mathrm{g}(x, y)$ as $\mathrm{f}(x, y) - \mathrm{g}(x, y) = 0$. These two equations are equivalent in the sense that they have the same graph. But the way that **implicitplot** writes the equation is essential for how **implicitplot** works. So as far as **implicitplot** is concerned, we can say that all equations are of the form $\mathrm{F}(x, y) = 0$, where we are using $\mathrm{F}(x, y)$ to represent any expression in x and y.

We have to give the **implicitplot** command two ranges, one for the x variable and one for the y

variable. Let us denote these ranges by $x = a .. b$ and $y = c .. d$, where a, b, c, d are any four real numbers with $a < b$ and $c < d$. The `implicitplot` command only tries to graph the equation $\mathrm{F}(x, y) = 0$ within the rectangle determined by the four points (a, c), (b, c), (b, d) and (a, d). Let us recall exactly what it is that `implicitplot` is trying to graph. Within the rectangle determined by (a, c), (b, c), (b, d) and (a, d), `implicitplot` is looking for every ordered pair (x, y) that solves the equation $\mathrm{F}(x, y) = 0$. Since the solution of the equation $\mathrm{F}(x, y) = 0$ is in general a curve, there will usually be an infinite number of points in the rectangle that solve the equation. So in fact the `implicitplot` command does not look for every point that solves the equation, just enought of them so that it can connect the points with straight line segments and get a good approxmation of the solution curve. But notice right away that there is something very different here from what the `plot` command does. The `implicitplot` command has to *search* through the rectangle to find points that it should plot. The `plot` command *computes* the points that it should plot. Given any value for the independent variable, the `plot` command has a very specific way to generate a point to plot; it plugs the indepdent variable into the function to get the value of the dependent variable. The `implicitplot` command does not have a way to automatically generate the points that it should plot. The `implicitplot` command needs a way to search through the rectangle and find points on the solution curve.

Here is how the `implicitplot` command finds points that it should plot. The first step is to compute a grid of sample points within the rectangle determined by the points (a, c), (b, c), (b, d) and (a, d). To get the grid points, the intervals from `a` to `b` and `c` to `d` are each divided into 25 subintervals of size $\Delta x = (b - a)/25$ and $\Delta y = (d - c)/25$, respectively, and then the grid points have coordinates $(x_i, y_j) = (a + i\,\Delta x, c + j\,\Delta y)$ for i from 0 to 25 and j from 0 to 25 (so the grid has a total of 676 points). The second step is to evaluate the expression $\mathrm{F}(x, y)$ at all 676 grid points (x_i, y_j). The third step is to look for pairs of adjacent grid points that have opposite signs. By **adjacent grid points** we mean the following, for any i between 1 and 25 and any j between 0 and 24, the grid points adjacent to (x_i, y_j) are (x_i, y_{j+1}), (x_{i-1}, y_j) and (x_{i-1}, y_{j+1}), that is, the three grid points just north, west, and northwest of (x_i, y_j). If a pair of adjacent grid points with opposite signs is found, that means that there is a point from the curve somewhere on the line segment joining the two grid points. (Why is the last statement true? What important assumption are we making about the expression $\mathrm{F}(x, y)$?). The fourth step is to compute approximate points on the curve by using linear interpolation between the adjacent grid points that have opposite signs. For example, if at the adjacent grid points (x_i, y_j) and (x_i, y_{j+1}) the expression $\mathrm{F}(x, y)$ has values z_j and z_{j+1} (where the numbers z_j and z_{j+1} have opposite signs), then the `impliciplot` command will plot the point with coordinates

$$(x_j, \frac{z_j\, y_{j+1} - y_j\, z_{j+1}}{z_j - z_{j+1}}).$$

These approximation points are the points that are connected by line segments to create the curve $\mathrm{F}(x, y) = 0$.

Exercise: What point is ploted if one of z_j or z_{j+1} is zero? What should be done if both of z_j and z_{j+1} are zero?

```
>
```

Exercise: Derive the linear interpolation formula used just above.

```
>
```

Exercise: Assuming that z_j and z_{j+1} have opposite signs, prove that $\frac{z_j y_{j+1} - y_j z_{j+1}}{z_j - z_{j+1}}$ lies between y_j and y_{j+1}. (Hint: Think in terms of weighted averages.)

```
>
```

Let us notice several things about the four step algorithm given above. First, the grid points (x_i, y_j) are not the points that `implicitplot` ends up plotting. The plotted points will usually lie between the grid points, unless one of the grid points lies right on the curve $\mathrm{F}(x, y) = 0$. In that case, `implicitplot` will not use the interpolation formula and it will plot the grid point. Second, the number of points used in the grid can be changed. The `implicitplot` command has a `grid` option and `grid=[m,n]` tells `implicitplot` to use a grid with `m` points in the *x* direction and `n` points in the *y* direction. Third, notice that the third step in the algorithm is the step where the searching needs to be done. All of the other steps are computed using specific formulas. This searching step can take quite a bit a time, as the next exercise demonstrates.

Exercise: With the default `grid=[26,26]`, so that there are 676 grid points, how many pairs of adjacent grid points need to be compared? If the grid is changed to `grid=[100,100]`, how many pairs of adjacent grid points need to be compared?

```
>
```

Exercise: The following command graphs the equation `0=0` over a rectangle. What should the graph look like? Explain why the graph looks as it does. (Hint: Recall the definition of adjacent points that we gave above.)

```
> implicitplot( 0=0, x=0..10, y=0..10, axes=none );
>
```

Let us look at a simple example of using `implicitplot`. We graph the equation $x^2 + y^2 = 1$ with `style=point`, so that we can see the points that `implicitplot` computes.

```
> implicitplot( x^2+y^2=1, x=-1..1, y=-1..1,
                style=point, symbol=circle, scaling=constrained
  );
```

Here is a trick that lets us graph the grid points that `implicitplot` used along with the points from the equation that `implicitplot` plotted. In the next command we graph both the equation `0=0` (which will produce the grid points for us) and the equation `x^2+y^2=1` using the options `style=point` and `grid=[11,11]` so that we can see the distinct points more clearly.

```
> implicitplot( {0=0, x^2+y^2=1}, x=-1..1, y=-1..1,
  grid=[11,11],
                    style=point, symbol=circle, scaling=constrained
  );
```

It would be nice to use different colors and symbols for the points from the grid and the equation. We cannot do that with a single `implicitplot` command. So the next example uses `implicitplot` twice and a `display` command to combine the two graphs.

```
> implicitplot( 0=0, x=-1..1, y=-1..1, color=black,
  grid=[11,11],
                   style=point, symbol=diamond, scaling=constrained
  ):
> implicitplot( x^2+y^2=1, x=-1..1, y=-1..1, color=red,
  grid=[11,11],
                   style=point, symbol=circle, scaling=constrained
  ):
> display( [%%, %] );
>
```

Exercise: For each red point in the last graph, identify which pair of adjacent (black) grid points was used to generate that point on the graph of the equation.

```
>
```

Exercise: The circle $x^2 + y^2 = 1$ is very symmetric, the square determined by the ranges `x=-1..1, y=-1..1` is also very symmetric, and the grid determined in this square region by the option `grid=[11,11]` is also symmetric. But the computed points on the graph of the equation `x^2+y^2=1` are not as symmetric as we might expect. Use the algorithm defined above to explain the loss of symmetry in the computed points.

```
>
```

Exercise: Repeat the last two exercises for the following graph of the equation $x^6 + y^6 = 1$.

```
> implicitplot( 0=0, x=-1..1, y=-1..1, color=black,
  grid=[11,11],
                   style=point, symbol=diamond, scaling=constrained
  ):
> implicitplot( x^6+y^6=1, x=-1..1, y=-1..1, color=red,
  grid=[11,11],
                   style=point, symbol=circle, scaling=constrained
  ):
> display( {%%, %} );
>
```

Exercise: Explain in detail the anomalies in the following two graphs (which are from the last worksheet) of the equation max($|x|,|y|$) = 1.

```
> implicitplot( max(abs(x),abs(y))=1, x=-2..2, y=-2..2,
                 scaling=constrained );
> implicitplot( max(abs(x),abs(y))=1, x=-1..1, y=-1..1,
                 scaling=constrained );
>
```

Exercise: Repeat the last exercise for the following graph of $|x|+|y|=1$. Can you explain the broken (dashed) lines on two of the edges?

```
> implicitplot( abs(x)+abs(y)=1, x=-1..1, y=-1..1,
                 scaling=constrained );
>
```

Now let us return to our first `implicitplot` example. Why is the following graph empty, even though we know that the graph of the equation is a circle of radius one centered at the origin?

```
> implicitplot( (x^2+y^2)*exp(1-(x^2+y^2))=1, x=-1..1, y=-1..1 );
```

To help answer this question, let us graph the expression `(x^2+y^2)*exp(1-(x^2+y^2))-1` that `implicitplot` is working with.

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

From the *z*-axis in the graph we notice that the expression `(x^2+y^2)*exp(1-(x^2+y^2))-1` is always negative except for where it is 0. In order for the `implicitplot` algorithm to plot a point, it needs to find adjacent grid points with opposite signs. But with an expression that is always negative, there can never be adjacent grid points with opposite signs. So unless one of the grid points happens to land right on the graph of the equation (which in not very likely, even with a lot of grid points), the `implicitplot` algorithm will not plot any points and we get an empty graph.

```
>
```

Exercise: Explain the results of the following commands. How are they different than the last `implicitplot` command?

```
> implicitplot( (x^2+y^2)*exp(1-(x^2+y^2))=.99, x=-1.5..1.5,
  y=-1.5..1.5,
                  scaling=constrained);
> implicitplot( (x^2+y^2)*exp(1-(x^2+y^2))=.999, x=-1.5..1.5,
  y=-1.5..1.5,
>                 scaling=constrained);
> implicitplot( (x^2+y^2)*exp(1-(x^2+y^2))=.999, x=-1.5..1.5,
  y=-1.5..1.5,
>                 grid=[100,100], scaling=constrained);
>
```

Exercise: Use the expression $r^4 - 18\,r^2 + 81$ to derive an expression $\mathrm{F}(x, y)$ in x and y such that the graph of $\mathrm{F}(x, y) = 0$ is a circle but `implicitplot` cannot draw the graph. Use the `solve` command to see if it can find analytic expressions for the solution of $\mathrm{F}(x, y) = 0$ (it should be able to). Give the equation $\mathrm{F}(x, y) = 0$ a suitable "pertubation" so that `implicitplot` can approximate the correct graph of the equation.

```
>
```

In general, the algorithm for `implicitplot` has problems when the curve it is trying to graph happens to be near critical points for the expression $\mathrm{F}(x, y)$. If the graph of the expression $\mathrm{F}(x, y)$ is locally flat near the curve $\mathrm{F}(x, y) = 0$, then it is difficult for the `implicitplot` algorithm to find the adjacent grid points with opposite signs that are essential for the algorithm to work. The next three exercises show other problems that `implicitplot` can have because of critical points.

```
>
```

Exercise: What should the graph of the equation $x^2 - y^2 = 0$ look like? Explain the little square box at the center of the following graph.

```
> implicitplot( x^2-y^2=0, x=-1..1, y=-1..1, axes=none );
>
```

Exercise: Explain in detail the cause of the difference between the following two graphs. Notice that the commands differ only in the `grid` option.

```
> implicitplot( cos(x)=1, x=-6*Pi..6*Pi, y=-6*Pi..6*Pi,
  grid=[25,25] );
> implicitplot( cos(x)=1, x=-6*Pi..6*Pi, y=-6*Pi..6*Pi,
  grid=[26,26] );
>
```

Here is an interesting case of `implicitplot` having trouble drawing a graph. The following graph is not very accurate.

```
> implicitplot( 2*x^4+y^4-3*x^2*y-2*y^3+y^2=0, x=-2..2, y=0..2.1
  );
```

We can increase the grid size and get a better graph (the following graph might take a couple of minutes to draw).

```
> implicitplot( 2*x^4+y^4-3*x^2*y-2*y^3+y^2=0, x=-2..2,
  y=0..2.1,
                     grid=[200,200], axes=none );
```

The last graph is still not complete. There is a hole in the graph at the bottom near the origin. Let us look at a graph of the expression `2*x^4+y^4-3*x^2*y-2*y^3+y^2` to see why the `implicitplot` command is having problems. The following command graphs this expression along with the *xy*-plane. Where the *xy*-plane cuts this surface is the curve `implicitplot` was trying to graphing.

```
> plot3d( {2*x^4+y^4-3*x^2*y-2*y^3+y^2, 0}, x=-2..2, y=0..2.1 );
```

Notice how the surface just barely passes through *xy*-plane. The next command stretches the scale on the *z*-axis so that we can "blow up" the detail of the shallow part of the graph. The graph is initially drawn looking staight down the *z*-axis, so that we can see a crude trace of the curve `inplicitplot` was trying to graph. Rotate the graph to see the detail of the shallow part. Notice just how shallow the "inner" bump of the graph is. Can you figure out what the real shape of this graph is?

```
> plot3d( {2*x^4+y^4-3*x^2*y-2*y^3+y^2, 0}, x=-2..2, y=-0.2..2.1,

            view=-0.1..0.1, grid=[50,50], orientation=[-90,0] );
```

It is possible to get a very good graph of the curve defined by the equation $2\,x^4+y^4-3\,x^2\,y-2\,y^3+y^2=0$. This is done by parameterizing the curve. For the details see *Introduction to Maple*, 3rd Edition, by A. Heck, page 428.

```
>
```

Exercise: Let us define the following function.

```
> f := exp(-(x-3)^2-y^2) - exp(-(x+3)^2-y^2);
```

Here is a graph of this function. Notice that it has one spike going up and one going down.

```
> plot3d( f, x=-4..4, y=-9..9 );
```

Here is a (two dimensional) contour diagram for this function. Why do you think that this diagram appears the way that it does?

```
> plots[contourplot]( f, x=-4..4, y=-9..9 );
>
```

Here are some examples of `implicitplot` using polar coordinates. The first two graphs show us what the default 26 by 26 point grid looks like in polar coordinates.

```
> implicitplot( 0=0, r=0..1, t=0..2*Pi, coords=polar,
  style=point,
>                                       scaling=constrained );
> implicitplot( 0=0, r=0..1, t=0..Pi, coords=polar, style=point,
>                                     scaling=constrained );
>
```

Exercise: Explain why the following grid looks the way that it does.

```
> implicitplot( 0=0, r=-1/2..1, t=-Pi/2..Pi/2, coords=polar,
  style=point,
>
  scaling=constrained);
>
```

```
> implicitplot( 0=0, r=0..1, t=0..2*Pi, color=black,
  coords=polar,                style=point, symbol=diamond,
  scaling=constrained ):
> implicitplot( r=cos(t), r= 0..1, t=0..2*Pi, color=red,
  coords=polar,                style=point, symbol=circle,
  scaling=constrained ):
> display( {%%, %} );
> 
> implicitplot( 0=0, r=0..1, t=0..Pi/2, color=black,
  coords=polar,                style=point, symbol=diamond,
  grid=[26,8], scaling=constrained ):
> implicitplot( r=cos(t), r= 0..1, t=0..Pi/2, color=red,
  coords=polar,                style=point, symbol=circle,
  grid=[26,8], scaling=constrained ):
> display( {%%, %} );
> 
> implicitplot( 0=0, r=0..1, t=0..Pi/2, color=black,
  style=point, symbol=diamond, grid=[26,8], scaling=constrained
  ):
> implicitplot( r=cos(t), r= 0..1, t=0..Pi/2, color=red,
  style=point, symbol=circle, grid=[26,8], scaling=constrained ):
> display( {%%, %} );
> 


> implicitplot( 0=0, r=0..1, t=0..2*Pi, color=black,
  coords=polar,                style=point, symbol=diamond,
  scaling=constrained ):
> implicitplot( r^2=cos(2*t), r= 0..1, t=0..2*Pi, color=red,
  coords=polar,                style=point, symbol=circle,
  scaling=constrained ):
> display( {%%, %} );
> 

> implicitplot( 0=0, r=0..1, t=0..Pi/2, color=black,
  coords=polar,                style=point, symbol=diamond,
  grid=[26,8], scaling=constrained ):
> implicitplot( r^2=cos(2*t), r= 0..1, t=0..Pi/2, color=red,
  coords=polar,                style=point, symbol=circle,
  grid=[26,8], scaling=constrained ):
> display( {%%, %} );
> 
```

```
>
```

6.4. The `plot3d` command

Let us now look more carefully at how Maple draws three dimensional graphs for functions of two variables. The basic idea here is still pretty much the same as with two dimensional graphs. To draw a graph of a function of two variables Maple simply plots points. But for functions of two variables Maple does not do any adaptive spacing of points. Given a function f(x, y) of two variables and a rectangular domain with x between a and b and y between c and d, Maple will, by default, chose a 25 point by 25 point grid of evenly spaced sample points within the rectangle formed by the four points (a, c), (b, c), (b, d), and (a, d). Maple will then evaluate f at the 625 sample points and plot these values. Here is a simple example. The following command plots a paraboloid using points and then views the plot looking straight down the z-axis. So the graph shows us the grid that Maple used for evaluating the function. If you look carefully you can see that the grid is evenly spaced and has 25 points on each edge. If you click on this grid and then use the mouse to rotate it, you can see the three dimensional aspect of the graph.

```
> plot3d( x^2+y^2, x=-4..4, y=-4..4, style=point,
  orientation=[0,0],
          color=black );
```

Here is another example, this time using a saddle surface.

```
> plot3d( x^2-y^2, x=-4..4, y=-4..4, style=point,
  orientation=[0,0],
          axes=frame, color=black );
```

As we mentioned just before, Maple does not attempt to do any adaptive spacing of points in the rectangular grid. A default grid of 25 points by 25 points (for a total of 625 sample points) is good enough for a rough graph of most functions, but as we will soon see, it often does not give very smooth graphs. We can control the number of points in the sample grid using the **`grid`** option to **`plot3d`**. The next example plots the saddle surface with a grid of 10 points by 15 points.

```
> plot3d( x^2-y^2, x=-4..4, y=-4..4, style=point, grid=[10,15],
          orientation=[0,0], color=black );
```

Of course, most of the time we will want to choose grid values that are larger than the default values, not smaller.

```
>
```

Once Maple has plotted all of the sample points for the graph, the points should somehow be connected together. There are two basic choices for connecting the points, using either line segments or "patches". First let us try line segments. The next two commands draw the paraboloid and the saddle using line segments to connect the sample points.

```
> plot3d( x^2+y^2, x=-4..4, y=-4..4, style=line, color=black );
> plot3d( x^2-y^2, x=-4..4, y=-4..4, style=line, color=black );
```

These kinds of graphs are also called **wireframe** graphs. The graph is, in a sense, made up of wires bent into the correct shape. Notice how in these two graphs you can "see through" the graphs since the spaces between the wires are considered empty. For some surfaces this see through aspect of a wireframe graph can be confusing. So there is a way to draw a wireframe graph that is not transparent. The next command graphs the saddle surface using the **hidden** option. Think of this as making the spaces between the wire frames opaque, so some of the wireframe will be hidden from view.

```
> plot3d( x^2-y^2, x=-4..4, y=-4..4, style=hidden, color=black );
```

Maple has two styles of wireframes that it can draw for a surface, **rectangular** and **triangular**. These styles are for an option that is confusingly called **gridstyle**. The name **gridstyle** is confusing because the **rectangular** and **triangular** styles do not in any way affect the rectangular grid of sample points used to graph the surface. These two styles just determine the way the sample points are connected by lines to make the wireframe. The next two commands draw the saddle surface with the rectangular and triangular wireframe styles (the rectangular wireframe is in fact the default). The surfaces are initially drawn looking straight down the *z*-axis, so that you can see how the two styles of wireframes get their names. Be sure to rotate both graphs and compare the appearance of each style.

```
> plot3d( x^2-y^2, x=-4..4, y=-4..4, style=hidden,
  gridstyle=rectangular,
           orientation=[0,0], color=black );
> plot3d( x^2-y^2, x=-4..4, y=-4..4, style=hidden,
  gridstyle=triangular,
           orientation=[0,0], color=black );
```

If you want to convince yourself that the option **gridstyle=triangular** has no affect on the rectangular grid of sample points for the graph, just change the **style** option from **hidden** to **point** in the last command.

```
>
```

Now let us connect the points with "patches" of surface using the **patch** option.

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

The **patch** option is in fact the default option, so we do not really need to include it. Here is the saddle surface graphed with patches connecting the sample points.

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

The **patch** option actually draws both the wireframe and the patches between all the sample points. This seems to give the most all around useful graph. It is also possible to graph just the patches, without the wireframe. (Notice that the option name is confusingly called **patchnogrid**, but of course there still is a grid. There is no wireframe.)

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

We have graphed the paraboloid and the saddle using five different styles, **point**, **line** (or **wireframe**), **hidden**, **patch**, and **patchnogrid**. Each of these five styles (plus two others) can be choosen in the **plot3d** command, or they can be choosen by clicking on a 3D graph and using buttons from the context bar at the top of the Maple window, or by right clicking on the 3D graph and using the pop-up context menu.

```
>
```

Now let us look at an example where the default grid of 25 by 25 samples points is not really enough. The function

$$f(x) = \sin(\sqrt{x^2 + y^2})$$

has a maximum value of 1 and a minimum value of -1. The function attains its maximum on circles of radius $\pi/2 + 2\,\pi\,n$ centered at the origin. So the following graph should have a nice flat circular top to it, but in fact the top edge is quite lumpy.

```
> f := sin(sqrt(x^2+y^2));
> plot3d( f, x=-4*Pi..4*Pi, y=-4*Pi..4*Pi );
```

To really see the lumpiness, use the context bar for the 3D graph to switch between the `pathnogrid` and `hidden` plot styles. Notice how the lumpiness exactly follows the grid lines from the wireframe. These grid lines are all the information that the `plot3d` command has about the function. To improve the graph we need the gridlines to be closer together. Let us try a higher value for the number of sample points.

```
> plot3d( f, x=-4*Pi..4*Pi, y=-4*Pi..4*Pi, grid=[75,75] );
```

Now the lumpiness along the top edge is almost gone, but the graph does not really look good with the wire frame showing since the wireframe is now so dense. Try looking at the graph using the `pathchnogrid` style.

```
>
```

In the last example, if you try smaller grid sizes you will see that the lumpiness is still pretty evident. It took a 9 fold increase in the number of sample points (from 625 to 5625) to really improve the graph. This is also a 9 fold increase in the amount of time and memory that Maple needs to use to draw the graph. This is typical of drawing surfaces. It takes quite a bit of work to draw accurate images of surfaces. The default grid size of 25 by 25 points is a good compromise between the accuracy of the image and the time and memory needed to draw the image.

```
>
```

Exercise: Maple can draw surfaces over nonrectangular regions, but it will still use essentially a rectangular grid of sample points. Use the next two commands to try and explain how the `plot3d` command determines its grid of sample points when it is graphing over a nonrectangular region. Try switching between the `point`, `hidden` and `patch` styles to help you figure out how the sample points were choosen (use the context bar). The first command draws the saddle surface over a nonrectangular region. The graph is initially drawn looking straight down the *z*-axis so that you can see the shape of the region and the grid used in the region.

```
> plot3d( x^2-y^2, x=-4..4, y=-sqrt(16+x^2)..sqrt(16+x^2),
          style=point, orientation=[0,0], axes=frame );
```

The next command draws the paraboloid over a circular region, which helps the graph look more paraboloid like.

```
> plot3d( x^2+y^2, x=-4..4, y=-sqrt(16-x^2)..sqrt(16-x^2),
          style=hidden, orientation=[0,0], axes=frame );
>
```

There are many ways to control the appearance of the image that Maple draws for a surface. In the next several examples we will discuss the `orientation`, `shading`, `color`, and `light` options to the `plot3d` command. None of these options can change the accuracy of a graph. A graph's accuracy is determined by the ranges used for the threee axes and the number of sample points used to draw the graph. The options that we will go over now are mainly used to control the aesthetic appearance of a graph, though they can also be used to highlight, or draw attention to, some detail of a graph.

The `orientation` option determines the point, caled the **viewpoint**, in three dimensional space from which the surface is viewed. The viewpoint is determined by two angles, called θ and ϕ, that are measured in degrees. These two angles are essentially the spherical coordinates of the viewpoint on a sphere centered at the origin and sufficiently large to enclose the whole surface being drawn. Like spherical coordinates, θ measures an angle in the *xy*-plane from the positive *x*-axis and ϕ measures an angle from the positive *z*-axis. For example, here is the saddle surface with orientation (θ, ϕ) = (45,90), which is looking straight into the "edge" of the *xy*-plane, "halfway" between the positive *x* and *y* axes.

```
> plot3d( x^2-y^2, x=-4..4, y=-4..4, orientation=[45,90],
  axes=framed );
```

If you click on the last graph, you will see the θ and ϕ coordinates in little boxes on the left side of the 3D context bar at the top of the Maple window. There are little up and down arrows next to these angles that you can click on to change their values. You can also click directly on either of the boxes that hold the angles, which puts the cursor in one of the boxes, and then you can change the angle to any value and the image will chage orientation immediately. And if you click on the graph with the mouse and rotate it, you can see the orientation angles continuously changing as you rotate the graph. By playing around with these three different ways of setting the orientation, you should be able to give yourself a good sense of what it means.

```
>
```

Exercise: The orientation angles θ and ϕ are not quite exactly the same thing as spherical coordinates for the viewpoint. For example, for a given fixed radius ρ, the angles (θ, ϕ) = (0,0) and (θ, ϕ) = (45,0) in spherical coordinates describe the same point on the sphere (why?). But the orientation angles (θ, ϕ) = (0,0) and (θ, ϕ) = (45,0) do not describe the same orientation. Use the following two commands to help you explain exactly how and why these two orientations differ. How would you describe these two orientations in words? For what other values of θ and ϕ will the orientation angles differ from the angles in spherical coordinates?

```
> plot3d( x^2-y^2, x=-4..4, y=-4..4, orientation=[0,0],
  axes=normal );
> plot3d( x^2-y^2, x=-4..4, y=-4..4, orientation=[45,0],
  axes=normal );
>
```

Exercise: The orientation angles (θ, φ) = (0,0) look straight down the *z*-axis at the *xy*-plane. But the *x* and *y* axes are not in their usual position for graphs of the *xy*-plane. What orientation angles would look straight down the *z*-axis and have the *xy*-plane in the usual position (i.e., positive *x*-axis to the right and positive *y*-axis pointing vertically)?

```
>
```

Here is a simple demonstration of orientation angles that uses an animation. The `seq` command creates 21 3D graphs. Each graph created by the `seq` command has a slightly different orientation. The `display` command combines the 21 graphs into an animation. Click on the first frame of the animation, when it is displayed, to get the "VCR" buttons in the context bar.

```
> i -> plot3d( x^2+y^2, x=-4..4, y=-4..4,
  orientation=[45,45+i*18] ):
> seq( %(i), i=0..20 ):
> plots[display]( [%], insequence=true );
```

Notice that in this animation it appears as if the surface is rotating. But in fact, the surface is really fixed and it is the viewpoint that is moving.

```
>
```

Now let us turn to the use of color when graphing surfaces. The use of color with the graphs of curves is very straight forward. A curve is given a solid color that is used along the whole curve. But for surfaces, the use of color is much more subtle. First of all, if you look back at any of our graphs of surfaces, you see that they do not have a solid, uniform color to them. The color varies all about a surface in a manner that helps make the surface easier to view. In general, color is used with surfaces as a way to help make the shape and detail of a surface easier to see. Color is also used as an aesthetic tool, to make the surfaces more pleasing and interesting to look at. For example, here is the saddle surface drawn in the solid color blue. This graph is not all that appealing.

```
> plot3d( x^2-y^2, x=-4..4, y=-sqrt(16+x^2)..sqrt(16+x^2),
          style=patch, color=blue );
```

If we redraw the graph in solid blue and without the grid lines, then the surface actually becomes very difficult to visualize. In the following graph, with many choices of orientation it becomes almost impossible to even see that the graph is a surface.

```
> plot3d( x^2-y^2, x=-4..4, y=-sqrt(16+x^2)..sqrt(16+x^2),
          style=patchnogrid, color=blue );
```

On the other hand, with the default coloring of a surface, even without the grid lines the graph is obviously a surface from almost any orientation.

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

These last few examples show that it really does matter how color is choosen for a surface so that the surface is easy to visualize and pleasing to look at.

```
>
```

The `plot3d` command has two options that control the use of color for surfaces, the `color` option and the `shading` option. First we will discuss the `shading` option, which is the easiest way to get a good coloring scheme for a surface. The `shading` option provides five predefined coloring schemes for surfaces. They are called `xyz`, `xy`, `z`, `zhue`, and `zgrayscale`. The default coloring scheme is `shading=xyz`, and this is the coloring scheme that we have been seeing in all of the surfaces we have drawn so far. In this scheme, the color assigned to a piece of a surface depends on all three of the coordinates of the piece. With `shading=xy`, the color assigned to a piece of a surface depends on the two horizontal coordinates of the piece. With `shading=z`, the color assigned to a piece of a surface depends only on the vertical coordinate of the piece. With `shading=zhue` and `shading=zgrayscale`, the color assigned to a piece of a surface also depends only on the vertical coordinate of the piece. With `zgrayscale` only shades of gray are used so the graph is drawn in "black and white". With `zhue` all the colors of the spectrum are used, from violet to red, with violet at the bottom (minimum) of the surface and red at the top (maximum) of the surface. The following two commands draw the paraboloid and the saddle surface with `shading=xy`. Try changing this option to see what the other shading styles look like. You can change this option by editing the commands, or by right clicking on the graphs and using the "Color" menu item from the pop-up context menu, or by clicking on a graph and using the Color menu in the main Maple menu bar at the very top of the Maple window. Also try different combinations of `shading` and `style` options

```
> plot3d( x^2+y^2, x=-4..4, y=-sqrt(16-x^2)..sqrt(16-x^2),
             shading=xy, style=patch );
> plot3d( x^2-y^2, x=-4..4, y=-sqrt(16+x^2)..sqrt(16+x^2),
             shading=xy, style=patch );
```

There is one other `shading` option, and that is `shading=none`. This option turns off the color shading of the surface and draws the surface all in white. We will see what this option is used for when we discuss the `plot3d` command's lighting options below.

```
>
```

The other `plot3d` option for coloring a surface is the `color` option. We have already seen how this option can be used to give a graph a solid, uniform color, like in the `plot` command. But we have also seen that this is not a very useful way to color a surface. There is another way to use the `color` option and that is to use it with a **color function**. The simplest kind of color function is an expression, in the same variables as the function being graphed, that replaces the color name after the `color` option. The expression is used by the `plot3d` command to compute a color for each piece of the surface that depends on the horizontal coordinates of the piece (similar to the `shading=xy` option). The full spectrum of colors is used with the minimum value of the expression mapped to the color red and the maximum value of the expression mapped to the color violet (similar to the `shading=zhue` option). Here are a few simple examples. The next command graphs the constant function `1`, so the graph is a flat plane. The color function is the expression `x*y`. The initial orientation is looking straight down on the surface so that you can see how color depends on the `x` and `y` coordinates. Notice that in this graph the bands of color are shaped like hyperbolas. With the expression `x*y` as the color function, curves of constant color are hyperbolas (why?).

```
> plot3d( 1, x=0..10, y=0..10, color=x*y, style=patchnogrid,
               orientation=[0,0] );
```

The next example uses `x+y` as the color function. Now the curves of constant color are sloping lines.

```
> plot3d( 1, x=0..10, y=0..10, color=x+y, style=patchnogrid,
               orientation=[0,0] );
```

The next example uses `sin(x)*sin(y)` as the color function. Now the coloring is periodic in both directions.

```
> plot3d( 1, x=0..10, y=0..10, color=sin(x)*sin(y),
  style=patchnogrid,
               orientation=[0,0] );
> 
```

Exercise: The following command uses the color function `x*y` on the flat plane $z = 1$, as in the first example above, but now the graph is over the domain `x=-10..10, y=-10..10`. The shading seems to be quite a bit different in this example compared to the first example. Explain in detail how this shading was determined. (Hint: Where are the minimum and maximum of `x*y` in this domain? What is the order of colors from red (at the min) to violet (at the max)? What color is used half way between the min and max and where is that color in this shading?)

```
> plot3d( 1, x=-10..10, y=-10..10, color=x*y, style=patchnogrid,

               orientation=[0,0] );
> 
```

Here is a possible interpretation and use for a color function. Imagine that the plane $z = 1$ is a sheet of metal that has nonuniform temperature and that the function $\mathrm{T}(x, y) = xy$ represents the temperature of the metal at the point with coordinates (x, y). Then the graph of the plane $z = 1$ with color function `x*y` will not only graph the surface but it will also graph the temperature information by translating temperatures into colors. The next two commands graph the paraboloid and the saddle surface with the color function `x*y`. In each example, the function being graphed (`x^2+y^2` and `x^2-y^2` repectively) determines the shape of the surface, and the color function (`x*y`) can be interpreted as giving the temperature at a point on the surface. So each of the following two graphs can be thought of as graphing two functions simultaneously. (Try modifying the color function used in these graphs to see how that affects the appearance of the surface.)

```
> plot3d( x^2+y^2, x=-4..4, y=-4..4, color=x*y,
  orientation=[0,0],
               style=patchnogrid );
> plot3d( x^2-y^2, x=-4..4, y=-4..4, color=x*y,
  orientation=[0,0],
               style=patchnogrid );
```

So a color function can be used to add more information to the graph of a surface. There are many quantities that a color function could represent besides temperature (for example density, thickness, slope, curvature, etc.).

```
>
```

Now let us look at a more complex kind of color function. Instead of having a single expression for the color at a point on the surface, we can have three expressions, one each for the amount of red, green, and blue color that will be mixed together to form the composite color of the surface at a point. Here are two examples, one using the paraboloid and one using the saddle surface.

```
> plot3d( x^2+y^2, x=-4..4, y=-4..4, color=[x,y,x*y],
           orientation=[0,0], style=patchnogrid );
> plot3d( x^2-y^2, x=-4..4, y=-4..4,
  color=[sin(x),sin(y),cos(x*y)],
           orientation=[0,0], style=patchnogrid );
```

Notice that, in a certain sense, the last two graphs are simultaneously graphing four function worth of information. But these kinds of color functions are pretty difficult to interpret and they are mostly used in specialized situtations.

```
>
```

Now let us turn to the lighting options of the `plot3d` command. There are three option related to lighting, `light`, `ambientlight`, and `lightmodel`. These options simulate having lights shining on a surface. These lighting options are closely related to the `shading` and `color` options. The `shading` and `color` option provides a way to give a surface a color scheme. The lighting options provides a way to have ambient or direct light shine on a surface. What makes the lighting options related to the coloring options is that the light that can shine on a surface will itself have a color, and so a surface can be colored by the light that shines on it.

Here is a simple example. The following command draws the saddle surface with two lights shining on it, one above the surface and one below the surface. Each of the two lights is determined by one of the two `light` options. Each light option has five parameters, two angles, ϕ and θ, that are the spherical coordinates of the location of the light source (but notice the change in the order of the angles!) and three numbers between 0 and 1 that give the proportion of red, gren, and blue light in the light source. The light source above the saddle is green light, and the light source below the saddle is yellow light. Notice that we also use the option `shading=none`, so that the saddle surface itself is uncolored, that is, it is white.

```
> plot3d( x^2-y^2, x=-4..4, y=-sqrt(16+x^2)..sqrt(16+x^2),
           light=[0,0,0,1,0], light=[180,0,1,1,0],
           shading=none, style=patchnogrid, grid=[50,50] );
>
```

Notice several things in this example. First, as you rotate the surface the lights do not seem to move. The green light shines from above and the yellow light shines from below. Second, notice that by using color from lights we can have a surface with different colors on different sides of the surface. When we used the `shading` and `color` options, the color at any point of the surface was always the same on both "sides" of the surface. Third, notice that the intensity, or brightness, of the light on the surface is related to the angle that the light makes with the surface. The surface is at its brightest wherever the surface is perpendicular to a light source (that is, wherever the normal vector of the surface points to a light source). The surface becomes darker as the angle the surface makes with the light beams decreases (that is, as the normal vector becomes more perpendicular to the light rays, the surface becomes darker). Having the surface become so dark in some spots may not be very desirable. Maple allows us to shine "ambient" light on a surface. This is light that comes from all directions. The next commands adds a bit of ambient white light. White light is made up of equal amounts or red, green, and blue light. So the option `ambientlight=[.5,.5,.5]` gives us a not too bright ambient white light that eliminates much of the dark shading that is in the last example.

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

  light=[0,0,0,1,0],light=[180,0,1,1,0],ambientlight=[.5,.5,.5],
          shading=none, style=patchnogrid, grid=[50,50] );
>
```

To see that the surface itself does not have any color in these examples, let us shine bright white light from both the top and the bottom.

```
> plot3d( x^2-y^2, x=-4..4, y=-sqrt(16+x^2)..sqrt(16+x^2),
          light=[0,0,1,1,1], light=[180,0,1,1,1],
          shading=none, style=patchnogrid, grid=[50,50] );
```

We can mix surface color with color from lighting, but the results can be confusing, unless you understand color and light. For example, the next command shines yellow light on a magenta surface, and we end up seeing red. This is because the definition of a surface being magenta is that it absorbs green light and reflects red and blue light. And yellow light is made of equal parts red and green light. So when the yellow light shines on the magenta surface, the green component of the yellow light is absorbed by the surface and only the red component of the yellow light is reflected by the surface into our eyes. (To see the true color of the surface, change the lights to white light.)

```
> plot3d( x^2-y^2, x=-4..4, y=-sqrt(16+x^2)..sqrt(16+x^2),
          light=[0,0,1,1,0], light=[180,0,1,1,0],
          color=magenta, style=patchnogrid, grid=[50,50] );
```

Here is another example. This is a red surface with green and blue lights shining on it, but we see only black. Why?

```
> plot3d( x^2-y^2, x=-4..4, y=-sqrt(16+x^2)..sqrt(16+x^2),
          light=[0,0,0,1,0], light=[180,0,0,0,1],
          color=red, style=patchnogrid, grid=[50,50] );
>
```

Choosing lighting options can be quite difficult. Maple provides four predefined lighting models, each of which has an interesting combination of direct and ambient lighting. We can choose one of these lighting models by using the `lightmodel` option. The models are called `light1`, `light2`, `light3` and `light4`. The following command uses the `light1` lighting model with no surface color, so the only color comes from the lighting scheme. Try the other light models and then try the different light models combined with different shading schemes. You can make all of the changes directly in the following command or you can right click on the graph and choose the "Color" menu item from the pop-up context menu and change both the lighting model and the shading style from this menu.

```
> plot3d( x^2-y^2, x=-4..4, y=-sqrt(16+x^2)..sqrt(16+x^2),
          lightmodel=light1, shading=none, style=patchnogrid );
> 
```

The use of lighting and color in the graphing of surfaces is really a part of the subject of computer graphics. Maple's abilities with light and color are not as sophisticated as a dedicated 3D computer graphics program. For example, the light sources in Maple do not cause shadows to be cast. But Maple's color and lighting options are easier to use than most 3D computer graphics programs. Using Maple's lighting and color options is a good way to experiment with these ideas and learn about basic 3D graphics. If you want to learn more about what Maple can do (for example, how to get Maple surfaces to cast shadows) then look at the book *Discovering Curves and Surfaces with Maple*, by G. Klimek and M. Klimek, Springer-Verlag, 1997.

```
> 

> 
```

6.5. Animations

A fun way to practice working with graphics is to work with animations. It is fairly easy to create interesting animations out of curves and surfaces. The following subsections explain the basics of using the `animate`, `animatecurve`, and `animate3d` commands, and also how to create animations using plot valued functions with the `seq` and `display` commands.

```
> 
```

6.5.1. Animating curves

Recall that an animation is made by creating a sequence of graphs, called **frames**, and then displaying the frames very quickly in the order that they were created in. There are several commands in Maple that can be used to create animations of parametric curves. The two simplest commands are `animate` and `animatecurve`. Each command produces a slightly different kind of animation.

```
> 
```

In the `animatecurve` command, each frame of the animation will graph the exact same function, but the range of the function will change from frame to frame. Here is a simple example. The following `animatecurve` command will draw 16 graphs of the function `sin(x)` (16 is the default number of frames) and the graph of the *i*'th frame will have the range $0 .. \frac{i\,2\,\pi}{15}$ where *i* goes from 0 to 15.

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

You can view the individual frames of the animation by "single stepping" through the frames using the appropriate button on the animation context bar.

In general, a command of the form

```
animatecurve( f(x), x=a..b, frames=n )
```

will create n graphs of the function f(x) with the i'th graph having the range a .. $a+\frac{i\,(b-a)}{n-1}$ for i from 0 to $n-1$. The n graphs are used as the frames of the animation.

You can also animate more than one function at a time by putting the functions inside of a pair of braces.

```
> plots[animatecurve]( {sin(x), cos(x)}, x=0..2*Pi, frames=50
  );
>
```

In the `animate` command, each frame of the animation draws a function over the same range as in every other frame. But each frame has a parameter value associated to it and if the function being graphed depends on the parameter, then each frame will contain a different graph. Here is a simple example. Notice that in the following command, the function being graphed, $a\sin(x)$, has a parameter in it, a (for amplitude). The following `animate` command will draw 16 graphs (the default number of frames), the i'th graph will be of the function $\left(1+\frac{i}{15}\right)\sin(x)$ for i from 0 to 15, and each graph will have the range 0 .. 2 π.

```
> plots[animate]( a*sin(x), x=0..2*Pi, a=1..2 );
```

In general, a command of the form

```
animate( f(x,t), x=a..b, t=c..d, frames=n )
```

will create n graphs with the i'th graph being the graph of the function $\mathrm{f}\left(x, c+\frac{i\,(d-c)}{n-1}\right)$ for i from 0 to $n-1$, and each graph will have the range $x=a$ .. b. The n graphs are used as the frames of the animation.

You can also animate more than one function at a time by putting the functions inside of a pair of braces.

```
> plots[animate]( {a*sin(x), (2-a)*cos(x)}, x=0..2*Pi, a=1..2,
  frames=50 );
>
```

In the `animate` command, you cannot change the range used in each frame. In the `animatecurve` command you cannot change the function being graphed in each frame. Of the two, `animate` is the more versatile command. There is much more that you can do with a family of functions that depend on a parameter than you can do with a single function with a varying domain. In fact, there is a way to use `animate` to duplicate the functionality of `aninmatecurve`. Let us redo the `animatecurve` example above using `animate`. There are two aspects to this trick. First, we need to draw the graph as a parametric curve. Second, we use the frame parameter to scale the independent variable in the parametric curve.

```
> plots[animate]( [t*x, sin(t*x), x=0..2*Pi], t=0..1 );
```

Notice that the following, rather obvious trick, does not work.

```
> plots[animate]( sin(x), x=0..t*2*Pi, t=0..1 );
>
```

If you want to create an animation that, from frame to frame, changes both the function being graphed and the domain of the graph, then you need to use a more sophisticated technique for creating the animation. The more sophisticated technique makes use of "plot valued functions", `seq`, and `display`. This technique was described briefly in Section 4.11 from the previous worksheet.

Here is an example that contrasts `animate` and `animatecurve` and then shows how to use the more sophisticated technique for creating animations. The following animation draws spirals that grow in "radius", but each spiral always does exactly two revolutions.

```
> plots[animate]( [ r*t*cos(t), r*t*sin(t), t=0..4*Pi ], r=1..2
  );
```

The following animation draws a spiral spiraling out from the origin, but the "radius" of the spiral is fixed.

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

The next animation manages to use `animate` to combine the last two animations, so the animation has both the spirals spiraling out (i.e., the animation changes the domain of each spiral) and the "radius" of the spirals is growing (i.e., each frame draws a slightly different spiral). But this way of using `animate` results in an overly cryptic command.

```
> plots[animate]( [ (1+s)*(s*t)*cos(s*t), (1+s)*(s*t)*sin(s*t),
>                   t=0..4*Pi ], s=0..1 );
```

The following animation, using a combination of a plot valued function, `seq`, and `display` (see Section 10 from Worksheet 4), duplicates the last animation, but with a structure that is a bit easier to understand and work with.

```
> frames := s -> plot([ (1+s)*t*cos(t), (1+s)*t*sin(t),
  t=0..s*4*Pi ]);
> seq( frames(i/20), i=1..20 ):
> plots[display]( [%], insequence=true );
```

And with a plot valued function we can do even more. In the next example, the graph of our spiral goes through the colors of the rainbow as it spirals out (this cannot be done with the `animate` command since the `animate` cannot change any options to the `plot` command like `color`).

```
> frames := s -> plot([ (1+s)*t*cos(t), (1+s)*t*sin(t),
  t=0..s*4*Pi ],
>                       color=COLOR(HUE,s) ):
> seq( frames(i/60), i=1..60 ):
```

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

Here is an example of a simple animation that we can use to make another comparison between using the **animate** command vs. using a plot valued function. This is an animation of the parameterization of a circle where the two endpoints of the parameterized curve move away from a fixed starting point and meet at the point opposite to the starting point. Here is the animation using a plot valued function.

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

And here is the animation using the **animate** command. Decide for yourself which method you think is easier to understand and work with.

```
> plots[animate]( [cos(s*t), sin(s*t), t=-Pi..Pi], s=0..1,
  frames=60 );
>
```

Exercise: Create an animation of one circle moving around the the circumference of another circle.

```
>
```

Exercise: Create an animation of a line segment of length 2 π rolling itself up into a circle of radius one.

```
>

>
```

6.5.2. Animating surfaces

In the **animate3d** command, each frame of the animation draws a function over the same region as in every other frame. But each frame has a parameter value associated to it and if the function being graphed depends on the parameter, then each frame will contain a different graph. Here is a simple example where the function being graphed has an "amplitude" parameter **a**.

```
> plots[animate3d]( a*sin(sqrt(x^2+y^2)), x=-2*Pi..2*Pi,
  y=-2*Pi..2*Pi,
>                          a=0..1 );
```

In the next example, the function has a translation parameter **c** that makes the function move across the plane.

```
> plots[animate3d]( exp(-sqrt((x-c)^2+y^2)), x=-5..5, y=-5..5,
>                          c=-3..3 );
```

The next animation shows two traveling bumps crossing paths in the plane.

```
> plots[animate3d]( exp(-sqrt((x-c)^2+y^2)) +
  exp(-sqrt(x^2+(y-c)^2)),
>                      x=-10..10, y=-10..10,
>                      c=-10..10, frames=60, grid=[30,30] );
```
Here is a more elaborate animation of a bump travelling around the plane over a parametric path.
```
> g1 := plots[animate3d](
>
  exp(-sqrt((x-10*cos(2*c))^2+(y-10*sin(3*c))^2)),
>                      x=-15..15, y=-15..15,
>                      c=Pi/2..3*Pi/2, frames=60, grid=[40,40] ):
> g2 := plots[spacecurve]( [ 10*cos(2*t), 10*sin(3*t), 0],
>                             t=Pi/2..3*Pi/2, color=black ):
> plots[display](g1,g2);
>
```

Exercise: Make sure that you understand every part of the last example. Rewrite the example so that the component functions in the parametric path are called `f` and `g` and so that they can be easily changed. Try a few other paths.
```
>
```

Maple does not have an "animate3dsurface" command that would be analogous to `animatecurve`. But just as we were able to use `animate` to duplicate what `animatecurve` does, we can use `animate3d` to do what we might have expected from "animate3dsurface", that is, change the domain of the graph in each frame. Here is a simple example using a paraboloid. Notice that we must graph the function using a parametric graph and we have the frame parameter scale one of the independent variables of the function.
```
> plots[animate3d]( [a*x, y, (a*x)^2+y^2], x=-5..5, y=-5..5,
  a=0..1 );
```
Here is a similar example.
```
> plots[animate3d]( [a*x, y, 1/(1+(a*x)^2+y^2)],
>                      x=-3..3, y=-3..3, a=0..1, frames=60,
  axes=box );
```
Notice that in the last two examples, the domain was "unfolded" along the *x*-axis starting from the origin and moving out from the origin in both the positive and negative directions. The next example unfolds along the *x*-axis but starting at $x = -3$ and moving towards $x = 3$. This ends up looking a bit more natural, but it is harder to do using `animate3d`.
```
> plots[animate3d]( [-3+a*t, y, 1/(1+(-3+a*t)^2+y^2)],
>                      t=0..6, y=-3..3, a=0..1, frames=60,
  axes=box );
```
The next example unfolds the domain in both the *x* and *y* directions starting from the corner (*x*, *y*)=(-3,-3) towards the corner (*x*, *y*)=(3,3). In this example we have defined the function that we are graphing as a (anonymous) Maple function in order to make the `animate3d` command easier to read.

```
> (x,y) -> 1/(1+x^2+y^2):
> plots[animate3d]( [ -3+a*t, -3+a*s, %(-3+a*t, -3+a*s) ],
>                   t=0..6, s=0..6, a=0..1, frames=60, axes=box
  );
>
```

The last two examples were a bit awkward to write using the **animate3d** command. This command is not very good at unfolding a graph along its domain. Here is a better way to do the last two examples, using a plot valued function (actually a "plot3d valued function") along with **seq** and **dispaly** and the **insequence=true** option. Notice that in these examples it is much more obvious that we are unfolding the domain of the graph.

```
> t -> plot3d( 1/(1+x^2+y^2), x=-3..t, y=-3..3 ):
> seq( %(-3+i/10), i=1..60 ):
> plots[display]( [%], insequence=true, axes=box );
```

And notice how this next example needs only a slight change from the previous one.

```
> t -> plot3d( 1/(1+x^2+y^2), x=-3..t, y=-3..t ):
> seq( %(-3+i/10), i=1..60 ):
> plots[display]( [%], insequence=true, axes=box );
>
```

Now let us turn to parametric surfaces and use animations to unfold one parameter at a time in a parameterization. For an example, let us return to the standard parameterization of the sphere.

$$(\theta, \phi) \to [\sin(\phi)\cos(\theta), \sin(\phi)\sin(\theta), \cos(\phi)]$$

```
> [ sin(phi)*cos(theta), sin(phi)*sin(theta), cos(phi) ];
> plot3d( %, theta=0..2*Pi, phi=0..Pi, title="Sphere" );
```

As we mentioned earlier, 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 the *xy*-plane. The radius starts out at 0 when ϕ is 0 (at the "north pole"), and the radius then grows to 1 (at the equator) and then shrinks back to 0 (at the "south pole") as ϕ goes from 0 to π. The height above the *xy*-plane of the circle that is being drawn is given by the $\cos(\phi)$ term, which starts at 1 and decreases to -1.

```
>
```

Let us animate this parameterization by unfolding the parameterization in each of the θ and ϕ "directions". We can use either the **animate3d** command for these animations or we can use the plot valued function technique. We will give the examples both ways. This first animation shows the rotation caused by the θ parameter. 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=100, orientation=[-60,60],
>                   title="Animated Sphere");
```

Here is the same animation using a plot valued function. Notice that the input to the `plot3d` valued function is the range of one of the parameters of the parameterization.

```
> t -> plot3d( [ sin(phi)*cos(theta), sin(phi)*sin(theta),
  cos(phi) ],
>              theta=0..t, phi=0..Pi );
> seq( %(2*Pi*i/100), i=1..100 ):
> plots[display]( [%], insequence=true, orientation=[-60,60] );
```

The next animation shows how the ϕ parameter determines the radius 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=100, orientation=[30,100] );
```

The same animation done using a plot valued function.

```
> s -> plot3d( [ sin(phi)*cos(theta), sin(phi)*sin(theta),
  cos(phi) ],
>              theta=0..2*Pi, phi=0..s ):
> seq( %(Pi*i/100), i=1..100 ):
> plots[display]( [%], insequence=true, orientation=[30,100] );
>
```

The next animation shows how we can unfold both parameters at once.

```
> (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=100, orientation=[-60,90] );
```

The same animation done using a plot valued function. Notice how in this example, the plot valued function is a function of two variables.

```
> (t,s) -> plot3d( [sin(phi)*cos(theta), sin(phi)*sin(theta),
  cos(phi)],
>                  theta=0..t, phi=0..s );
> seq( %(2*Pi*i/100, Pi*i/100), i=1..100 ):
> plots[display]( [%], insequence=true, orientation=[-60,90] );
>
```

Exercise: In the last example, add a third input parameter to the plot valued function and use that parameter to change the radius of the horizontal circles (from, say, 1 to 3). Modify the `seq` command appropriately. (The surface should become more and more elliptical as it evolves.)

```
>
```

Let us create three similar animations for the standard parameterization of the torus. First, here is how we parameterize a torus.

$$(\theta, \phi) \to ((2+\cos(\phi))\cos(\theta), (2+\cos(\phi))\sin(\theta), \sin(\phi))$$

```
> [ (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" );
>
```

Recall that we have two ways of understanding this parameterization. The first way looks at this parameterization in a way similar to the parameterization of the sphere, that is, as a stack of horizontal circles whose radii are determined by the $2+\cos(\phi)$ term and whose heights are determined by the third component $\sin(\phi)$. The second way looks at the torus as a "circle's worth of circles". In this characterization, the terms $(2\cos(\theta), 2\sin(\theta), 0)$ determine a "circle of centers" in the *xy*-plane, the $(\cos(\theta)\cos(\phi), \sin(\theta)\cos(\phi), \sin(\phi))$ terms parameterize a vertical circle centered at the origin and parallel to a radial line from the "circle of centers", and then the torus parameterization is the sum $(2\cos(\theta), 2\sin(\theta), 0) + ($
$\cos(\theta)\cos(\phi), \sin(\theta)\cos(\phi), \sin(\phi))$. In this sum, choosing a value for θ chooses a point on the "circle of centers" and then the parameter ϕ parameterizes the circle from the "circle's worth of circles" with the center determined by θ.

The first animation unfolds the parameterization in the ϕ direction around the vertical circles.

```
> p := t -> plot3d( [(2+cos(phi))*cos(theta),
>                     (2+cos(phi))*sin(theta),
>                     sin(phi)],
>                    theta=0..2*Pi, phi=0..t );
> seq( p(2*Pi*i/100), i=1..100 ):
> plots[display]( [%], insequence=true,
>                      orientation=[90,130],
>                      scaling=constrained, title="Animated
  Torus" );
>
```

The next animation unfolds the parameterization in the θ direction around the circle of centers.

```
> p := s -> plot3d( [(2+cos(phi))*cos(theta),
>                     (2+cos(phi))*sin(theta),
>                     sin(phi)],
>                    theta=0..s, phi=0..2*Pi );
> seq( p(2*Pi*i/100), i=1..100 ):
> plots[display]( [%], insequence=true,
>                      orientation=[-90,60],
>                      scaling=constrained, title="Animated
  Torus" );
>
```

The third toral animation combines the previous two and it unfolds both of the parameters at the same time.

```
> p := (t,s) -> plot3d( [(2+cos(phi))*cos(theta),
>                         (2+cos(phi))*sin(theta),
>                         sin(phi)],
>                        theta=0..s, phi=0..t );
> seq( p(2*Pi*i/100, 2*Pi*i/100), i=1..100 ):
> plots[display]( [%], insequence=true,
>                      orientation=[-110,-160],
>                      scaling=constrained, title="Animated
  Torus" );
> 

> 
```

6.6. Defining coordinate systems

As we have seen many times, drawing graphs boils down to plotting points. Graphing commands do their job by plotting points and then possibly connecting those points together by line segments, or pieces of planes, in order to form curves or surfaces. But when we look more carefully at what graphing commands do, we see that what they really do is build certain data structures, PLOT data structures for 2-dimensional graphs and PLOT3D data structure for 3-dimensional graphs. One of the many data items contained in a PLOT (or PLOT3D) data structure is a list of pairs of numbers that represent the coordinates of the points that are to be plotted.

PLOT and PLOT3D data structures always contain a list of points in cartesian (or rectangular) coordinates (and the coordinates are computed using hardware floating point numbers). If a graphing command wants to work in some other coordinate system, after it computes the points it wants to graph in the other coordinate system and before it can build the PLOT or PLOT3D data structure, all of the points need their coordinates converted into cartesian coordinates.

Here is a simple example. The following command graphs a single point in polar coordinates.

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

Let us look at the PLOT data structure created by the last command.

```
> temp := %;
```

We see that the single point with polar coordinates [1, $\frac{\pi}{4}$] had its coordinates converted to cartesian coordinates with values [.7071067811865476, .7071067811865475] in the PLOT data structure. On the other hand, let us graph the single point with cartesian coordinates [1, $\frac{\pi}{4}$].

```
> plot( [[1, Pi/4]], coords=cartesian, style=point,
  symbol=diamond );
```

Let us look at the PLOT data structure for this last graph.

```
> temp := %;
```

Since the point we were plotting already was given by cartesian coordinates, no change of coordinates was needed.

Here is another example. Let us create a list of pairs of floating point numbers.

```
> point_list := [ seq( [evalhf(i/20*4*Pi),
                         evalhf(sin(i/20*4*Pi))], i=0..20) ];
```

Now ask `plot` to graph the list.

```
> plot( point_list );
```

By default, `plot` uses cartesian coordinates, so it did not apply any change of coordinates transformation to the list of pairs, as we can see from the next command which shows us the actual PLOT data stucture used.

```
> temp := %;
```

Now ask `plot` to once again graph the list, but this time it should interpret the pairs of numbers as pairs of polar coordinates.

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

If we look at the PLOT data structure used in the last graph, we see that the pairs of numbers from the original list were all transformed (from polar coordinates to rectangular coordinates).

```
> temp := %;
```

If we plot the same list of numbers again, using a still another coordinate system, we see that the original pairs of numbers are transformed in still a different way.

```
> plot( point_list, coords=cassinian );
> temp := %;
```

Here is a slight variation on the previous example. Let us start with the same list of pairs of numbers.

```
> point_list := [ seq( [evalhf(i/20*4*Pi),
                         evalhf(sin(i/20*4*Pi))], i=0..20) ];
```

Now let us perform our own transformation on the list of numbers to create a new list of numbers. Notice that the transformation uses the usual formuals for converting polar coordinates to rectangular coordinates.

```
> transformed_point_list := [seq( [p[1]*cos(p[2]),
                                   p[1]*sin(p[2])],
  p=point_list)];
```

Now `plot` the transformed list using rectangular coordinates. We get the same graph as when when we plot the original list using polar coordinates.

```
> plot( transformed_point_list );
>
```

Whenever we specify a coordinate system in a `plot` (or `plot3d`) command, what we are really doing is telling Maple what formulas it should apply to transform pairs of numbers into what will be interpreted as pairs of rectangular coordinates. For the predefined coordinates systems in Maple those transformations are built into Maple. But what is nice is that we can define our own coordinate systems for the `plot` (and `plot3d`) command by telling Maple what formulas to use in the transformation step of building the PLOT (or PLOT3D) data structure.

Here is a general idea of how we define a new coordinate system in Maple. Maple's internal plotting mechanism only knows how to plot points using cartesian coordinates. If we want to plot a point and we know its coordinates in some noncartesian coordinate system, we have to be able to tell Maple how to convert the noncartesian coordinates into cartesian coordinates so that Maple can figure out where to plot the point. For example, if we want Maple to plot the point with polar coordinates $(r, \theta) = (1, \pi)$, then Maple needs to know how to compute this point's cartesian coordinates, which are $(-1, 0)$. Knowledge of how to convert polar coordinates into cartesian coordinates has already been built into Maple (and soon we will see how). But in general, when we want to define a new coordinate system, one of the things we need to tell Maple is how to compute the cartesian coordinates of a point from the values of its new coordinates (and the other thing that we need to tell Maple is which of the two new coordinates will take on the preferred role of the independent variable when graphing functions).

We add coordinate systems to Maple, i.e., we specify a new coordinate transformation to use when building a PLOT (or PLOT3D) data structure, by using the `addcoords` command. To define a new coordinate system the `addcoords` command needs three parameters. The first parameter is the name we wish to give to our new coordinate system. (Be sure to choose a name that is not already in use.) The second parameter is a list of two variable names (that represent the variables in the new coordinates system). The second of these two names represents the coordinate that will be the independent variable when graphing a function. The third parameter is a list of the two formulas that convert our new coordinates into cartesian coordinates. Here is a description of the syntax of `addcoords`.

  `addcoords(` *name-of-new-coordinate-system*`,` **`[`***dependent-varaible*`,` *independent-variable* **`],`**
                  `[`
expression-for-horizontal-component-in-terms-of-independent-and-dependent-variables`,`

expression-for-vertical-component-in-terms-of-independent-and-dependent-variables`] )`

Let us look at some simple examples of using this command. First, let us define a new version of cartesian coordinates that stretches the vertical axis by a factor of two.

```
> addcoords( my_cartesian, [y,x],[x,2*y] );
```

To make it easier to figure out what is going on, we use the well known names x and y for the new coordinates. Notice that x is specified as the default independent variable (it is the second coordinate in the first list) and x is used directly as the horizontal coordinate in the final graph (that is, x is the first component in the second list). Also notice that, whatever value the new coordinate y has, two times that value is what is plotted in the vertical direction on the graph. So, for example, in this new coordinate system, the function $\sin(x)$ has a maximum of 2.

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

Now let us see how to define polar coordinates. We define our own version of polar coordinates that is exactly the same as the built in version.

```
> addcoords( my_polar, [r,theta], [r*cos(theta),r*sin(theta)] );
```

Notice that the variable θ is defined to be the default independent variable since it appears second in the first list. Here is a graph using this new coordinate system.

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

Notice that the graph is exactly the same if we use the built in polar coordinates.

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

Let us now create a polar coordinate system that has the radial coordinate as the default coordinate for the independent variable in a graph. All that we need do is modify the previous `addcoords` command so that the variable *r* appears second in the first list.

```
> addcoords( my_polar, [theta,r], [r*cos(theta),r*sin(theta)] );
```

Let us regraph the function $\sin(2\,x)$ (where now the variable *x* in this function represents the radial coordinate in the polar plane).

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

Recall that this is the same graph that we got in the last worksheet when we used parametric equations to draw a graph of $\theta = f(r)$ in polar coordinates. Here is what the graph looks like if we also plot this function over a negative range for the independent variable.

```
> plot( sin(2*x), x=-2*Pi..2*Pi, coords=my_polar );
>
```

Exercise: Explain why the following command draws the same graph as the second to last command.

```
> plots[implicitplot]( arctan(y/x)=sin(2*sqrt(x^2+y^2)),
                       x=0..6.4, y=-4.7..4, grid=[60,60] );
>
```

Exercise: Explain carefully why the following piece of graph is in the second quadrant.

```
> plot( sin(2*r), r=-Pi/2..0, coords=my_polar );
>
```

Exercise: Use the `addcoords` command to create several new cartesian coordinate systems that give the vertical axis the preferred role of the independent variable. Create coordinate systems with the positive coordinate direction going both up and down the vertical axis. Use your new coordinate systems to graph functions of the form $x = f(y)$ (where we will continue to label the horzontal axis as the *x*-axis and the vertical axis the *y*-axis). Also, create a cartesian coordinate system with the horizontal axis (the *x*-axis) as the independent variable, but with positive *x* coordinates going to the left.

```
>
```

Of the six possible graphs that we could make in cylindrical coordinates, there is one 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 $z = \mathrm{f}(x, y)$, and since (x, y) and (r, θ) both coordinatize the plane, then by analogy to rectangular coordinates it would seem reasonable for `plot3d` to graph $z = \mathrm{f}(r, \theta)$ when using cylindrical coordinates. Such graphs can in fact be very useful. For example, suppose we wanted to draw a graph of a function $\mathrm{f}(x, y)$ over the cardiod defined by $r = 1 + \cos(\theta)$. We might try to do this using `plot3d`'s ability to graph over nonrectangular domains in rectangular coordinates, but that would be difficult. What we would like to do is convert the function to cylindrical coordinates using $\mathrm{g}(r, \theta) = \mathrm{f}(r\cos(\theta), r\sin(\theta))$ and then draw a graph of $z = \mathrm{g}(r, \theta)$ 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` with cylindrical coordinates cannot graph functions of the form $z = \mathrm{g}(r, \theta)$. So we create a new version of cylindrical coordinates in which $z = \mathrm{g}(r, \theta)$ is the default graph.

```
> addcoords( my_cylindrical, [r,t,z], [r*cos(t),r*sin(t),z] );
```

Let us graph the piece of a paraboloid that is above a cardiod in the plane.

```
> plot3d( r^2, r=0..1+cos(t), t=0..2*Pi, coords=my_cylindrical );
```

Here is a verification that the above graph is correct.

```
> g1 := %:
> g2 := plot3d( x^2+y^2, x=-2..2, y=-sqrt(4-x^2)..sqrt(4-x^2),
                style=wireframe ):
> plots[display]( g1, g2 );
> 


> 
```