Sage for Undergraduates
Online Electronic-Only Appendices
About Color and 3D Plotting
(Unproofread Draft Copy)

Gregory V. Bard

Dept. of Mathematics, Statistics, and Computer Science, University of Wisconsin—Stout, Menomonie, Wi, 54751

E-mail address: bardg@uwstout.edu

Contents

How to Use these Appendices 1001

Appendix G. Color 2D Plotting 1003
G.1. Graphing in 2D with Color 1003
G.1.1. Overview 1003
G.1.2. Legends for Color 2D Plots 1005
G.2. Plotting Systems of Inequalities in 2D 1006
G.2.1. Plotting a Single Inequality 1007
G.2.2. Plotting a System of Inequalities, for Linear Programming 1008
G.2.3. Plotting Non-Linear Inequalities (2D Region Plots) 1013
G.3. Heat Maps, Contour Plots, and Density Plots 1014
G.3.1. Contour Plots vs Density Plots in Black-and-White 1014
G.3.2. Color Density Plots and Contour Plots 1016
G.3.3. Adding a Legend or Color Bar 1017
G.3.4. Other Color Maps 1018
G.3.5. Aspect Ratios in Contour Plots and Density Plots 1019
G.3.6. An Example from Mathematical Economics 1021
G.3.7. Dangerous Examples: The Pitfalls of Density Plots 1026
G.4. Saving your Image Files 1029
G.5. The Several Uses of the `show` Command 1030

Appendix H. Plotting in 3D 1035
H.1. Plotting $z = f(x, y)$ in Sage 1035
H.1.1. Brief Historical Background 1035
H.1.2. Using the New 3D Plotting Command 1036
H.1.3. Options for the New 3D Plot Command 1037
H.1.4. Using Color Maps to make Terrain Plots 1042
H.1.5. Comparing the Old and New Commands 1044
H.1.6. Using the old `plot3d` Command 1047
H.1.7. Table-Cloth Plots 1048
H.2. Plotting Implicit 3D Surfaces 1049
H.3. Plotting 3D Polyhedra 1050

H.3.1. Built-In Polyhedra, the Platonic Solids 1050
H.3.2. An Example: A Skeleton of an Octahedron 1051
H.3.3. The Polyhedron of a Linear Program 1052
H.4. The Best-Fit Plane 1054
H.5. Matrix Algebra and Intersecting Three Planes 1056
H.6. Plotting in Cylindrical Coordinates 1058
H.7. Plotting Volumes of Revolution in *Calculus II* 1061
H.8. Plotting in Spherical Coordinates 1065
H.8.1. Introduction to Spherical Coordinates 1065
H.8.2. Examples of Plotting with Spherical Coordinates 1067
H.9. 3D-Parametric Space Curves and Derivatives 1068
H.10. Plotting 3D Parametric Surfaces in Space 1072
H.11. 3D Vector Field Plots 1073
H.11.1. Example One: A Single Planet 1074
H.11.2. Example Two: Three Planets 1076
H.12. Functions of a Complex Variable 1076
H.13. The New Plotting Code 1076

Appendix I. Additional Index Entries 1081

How to Use these Appendices

These appendices are meant to extend the book *Sage for Undergraduates*, published by the American Mathematical Society in 2015. That 376-page book is available electronically for free on my webpage,

`www.gregorybard.com`

by clicking on "Books I've Written" or for purchase as a paperback at `www.ams.org` as well as at `www.amazon.com`.

Color images and three-dimensional images are not only visually stimulating, but they can help demonstrate a lot of important effects in the multivariarate calculus, the integral calculus, and other courses. Moreover, the sheer power of Sage in producing beautiful images (including in 3D) is one of Sage's most famous features.

The American Mathematical Society and I decided that *Sage for Undergraduates* should be printed in black and white, not in color, in order to keep the printed paperback as inexpensive as possible. This was to benefit those readers who are in economically challenged parts of the world where funding for education is limited (or US states, such as Wisconsin, where funding is available but withheld from universities for political reasons).

As a consequence of printing in black and white, 3D graphics could not be discussed in the book itself, because such images look like large amorphous blobs when printed in black and white. This obviously ruled out discussing color plotting in the printed book as well.

These online electronic-only appendices cover color plotting (Appendix G) and 3D plotting (Appendix H), and thereby render *Sage for Undergraduates* more complete. Moreover, I somehow left the very flexible and useful command `show` out of *Sage for Undergraduates*, and that is discussed in Section G.5 on Page 1030.

To help students learn, I will frequently challenge the reader with a task to perform after discussing some particular skill. Those "homework

problems" are marked with the bold-faced heading "A Challenge for You." While this was done in *Sage for Undergraduates*, the reader will find that it is done much more frequently in these appendices.

Prerequisites

Plotting in Sage was introduced in the book *Sage for Undergraduates* in Chapter 1.6, and many additional topics were presented in Chapter 3. You certainly do not need to read all of *Sage for Undergraduates* to use these appendices. However, you surely want to be familiar with Sage. I would recommend Sections 1.1, 1.2, 1.3, 1.4, 1.6, and 1.8, but mastery of Section 1.8 is not required.

All of computer algebra, but particularly the graphics aspects, will be best learned by tinkering. Therefore, I encourage you to "just mess around." As you experiment, some of the graphics that you produce will probably be fairly cool, so you might want to share them, and that is discussed in Section 1.13 of *Sage for Undergraduates*.

At many points in these appendices, I will discuss the 3D analog of a 2D technique. In such cases, I will alert the reader to the appropriate section of Chapter 3 in of *Sage for Undergraduates* where the 2D technique is explained. On the one hand, there certainly is no need to read all of Chapter 3 before starting to read these appendices. On the other hand, anyone who finds these appendices interesting will probably also find Chapter 3 of the main book interesting also.

Chapter 1 of of *Sage for Undergraduates* will get you familiar with SageMathCell (once called the Sage Single-Cell Server), but almost all readers who are interested in these appendices will want to quickly transition to `CoCalc.com`, formerly known as SageMathCloud. Videos are the best way to learn `CoCalc.com`, and those videos can be found at this URL:

`https://github.com/sagemathinc/cocalc/wiki/TalksAndVideos`

Appendix G

Color 2D Plotting

G.1. Graphing in 2D with Color

Color can make ordinary 2D graphs very attractive and visually appealing. It is not merely a question of adding clarity to an image, though that is often accomplished. There are many graphs which are easier understood in color than in black and white. Nonetheless, the bigger issue is about making mathematics beautiful. Many students have been inspired by the fascinating graphs of 2D and 3D structures in courses like *Calculus II* and *Calculus III*.

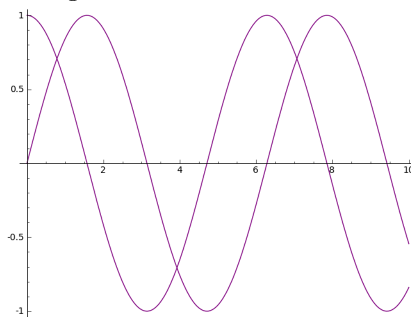
In any case, let us begin now, without further philosophical discussion.

G.1.1. Overview

An example of graphing with colors would be

```
plot([sin(x),cos(x)], 0, 10, color='purple')
```

producing



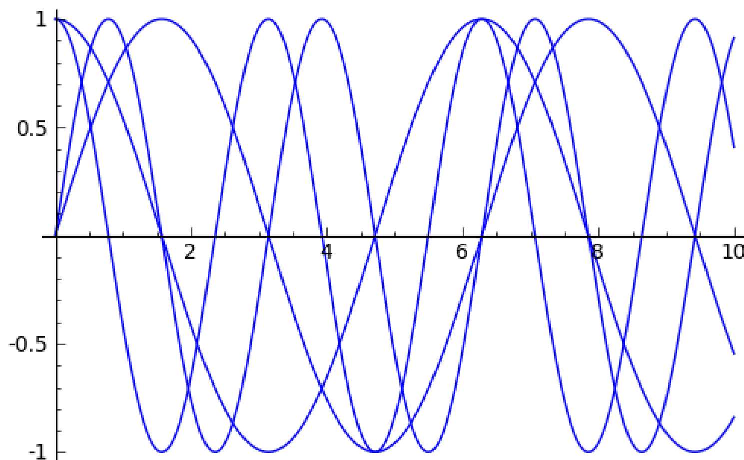
which looks like DNA to me. As you can see, you can plot multiple functions at the same time on the same graph. To do that, the list of functions should be separated by commas and enclosed in brackets.

The symbols `[sin(x), cos(x)]` are an example of a list in Sage. You can enclose any data with `[` and `]`, separating the entries with commas, to make a list. We've seen many examples of this syntax throughout this book, a notation which Sage inherited from the computer language Python.

However, it is important to not get carried away. Rarely does it make sense for four functions to appear together in one graph. For example,

```
plot([sin(x),cos(x),sin(2*x),cos(2*x)], 0, 10)
```

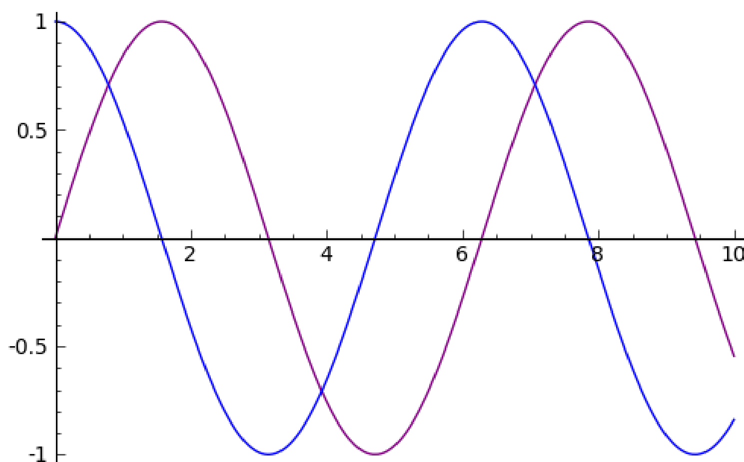
makes a total mess, as you can see



To plot multiple functions in multiple colors, the command is actually to add the plots:

```
plot(sin(x), 0, 10, color='purple') + plot(cos(x), 0, 10, color='blue')
```

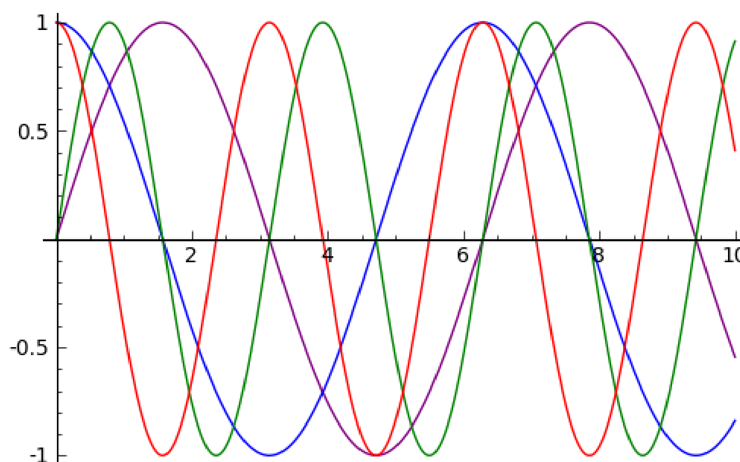
which produces something quite readable:



Or perhaps

```
plot(sin(x), 0, 10, color='purple')
         + plot(cos(x), 0, 10, color='blue')
         + plot(sin(2*x), 0, 10, color='green')
         + plot(cos(2*x), 0, 10, color='red')
```

As you can see below, that code produces a plot which is not readable but rather pretty.



Which Plot Gets to be on Top?

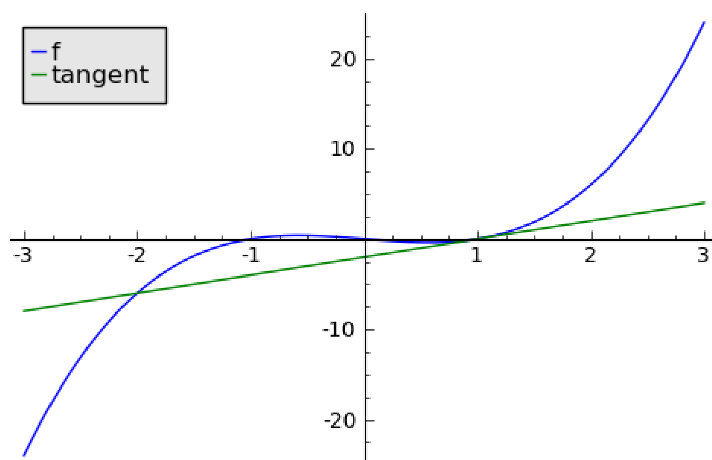
Once in a while, you might be concerned with which color goes on top for a set of plots that are superimposed. The criterion in Sage is simple—if two or more curves overlap at any point, the one given last in the sequence of plots (furthest to the right) goes on top. The one given first (furthest to the left) has to be on the bottom. Sometimes this is important in getting the graph to look precisely as you'd like it to.

G.1.2. Legends for Color 2D Plots

Sometimes when you have several functions in the same graph, it is nice to label them with a legend. Consider the example of plotting $f(x) = x^3 - x$ and the tangent line at $x = 1$, using the following code:

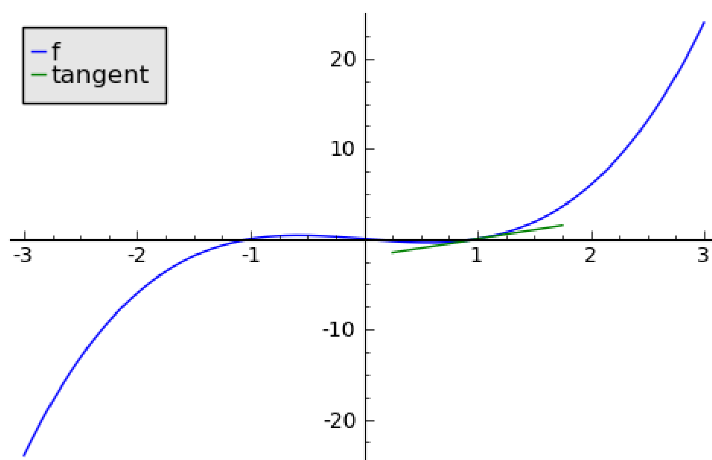
```
plot(x^3-x,-3,3,color='blue',legend_label="f") + plot(2*x-2,-3,3,
    color='green', legend_label="tangent")
```

which produces



You can also make the tangent line really short, if you like:

```
plot(x^3-x,-3,3,color='blue',legend_label="f") + plot(
    2*x-2,0.25,1.75,color='green',legend_label="tangent")
```



That last bit of code requires an explanation. The idea is that we told Sage to draw the curve $x^3 - x$ using the domain $x = -3$ to $x = 3$, while the line $2x - 2$ was to be drawn from $x = 0.25$ to $x = 1.75$. Since we gave the line a smaller domain, it appears shorter in the graph.

G.2. Plotting Systems of Inequalities in 2D

When we plot inequalities in x and y, instead of lines and curves we get shaded regions. These plots can be visually attractive, and are important in a diverse array of courses—ranging from *Elementary Algebra* up to doctoral-level coursework in *Operations Research*, *Industrial Engineering*, or *Linear Programming*.

G.2.1. Plotting a Single Inequality

Before we learn how to plot systems of inequalities, we should first learn how to plot a single inequality. Mercifully, this is extremely straightforward.

There are two ways to graph an inequality in the (x, y) plane. You can either shade the points that satisfy the inequality, or you can shade the points that violate the inequality. Books from *Elementary Algebra* to *College Algebra* tend to shade the points that satisfy the inequality. Those who do research in this area or teach courses in *Operations Research*, *Industrial Engineering*, or *Linear Programming* shade the points that violate the inequality, even in 100-level courses about that topic. Why is this the case? If you read the next subsection, where we plot systems of linear inequalities, you'll find out.

To graph an inequality such as

$$y \leq 1 - x^2$$

we shall instead plot the curve

$$y = 1 - x^2$$

and shade. Let's restrict the graphing window to $-2 \leq x \leq 2$, and $-2 \leq y \leq 2$.

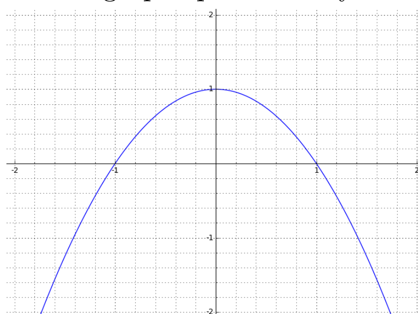
Instead of the simple command to plot a parabola

```
plot( 1-x^2, (x, -2, 2), ymin=-2, ymax=2, gridlines='minor' )
```

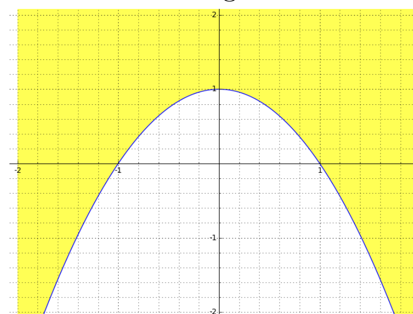
We're going to type instead

```
plot( 1-x^2, (x, -2, 2), ymin=-2, ymax=2, gridlines='minor',
           fill=10, fillcolor='yellow', fillalpha=2/3 )
```

The two graphs produced by the above commands are given below:



$y = 1 - x^2$



$y \leq 1 - x^2$

Note: In the graph on the right, points shaded yellow violate the inequality, while unshaded points and blue points satisfy the inequality.

We have three new optional parameters to explain now. The `fillcolor` parameter, as you might guess, identifies the color for filling in the region. The parameter `fill=10` means that we will fill from $y = 10$ until the parabola. In other words, we are filling above the parabola. If you

wanted to fill below the parabola, you would write `fill=-10`. The parameter `fillalpha=2/3` represents how transparent (or not) you would like the filling to be. This parameter must satisfy $0 \leq \alpha \leq 1$, where $\alpha = 0$ would specify an entirely transparent filling (making the filling invisible), and $\alpha = 1$ would specify a filling that is solid and opaque. However, even with $\alpha = 1$ the gridlines will show through. If you select $\alpha = 0$, the plot does not look shaded at all.

G.2.2. Plotting a System of Inequalities, for Linear Programming

It turns out that plotting a system of inequalities is only slightly more work than plotting a single inequality. The process is easier to explain via an example.

First Attempt

Suppose we want to plot the following system of linear inequalities.

$$\begin{aligned} y &\leq 1.5 - 2x \\ y &\leq 1 - x \\ y &\geq 0.5 - 0.5x \\ x &\geq 0 \\ y &\geq 0 \end{aligned}$$

Recall that in a system of inequalities, a point is only considered feasible if it satisfies each and every inequality. If even one inequality is violated by a point, then that point is not feasible. Students sometimes call this "the mother-in-law" property: if your mother-in-law gives you seven things to do, and you do only six of those, then you are in trouble because of the one that you didn't do.

To draw this system of inequalities, what we're going to do is draw the lines

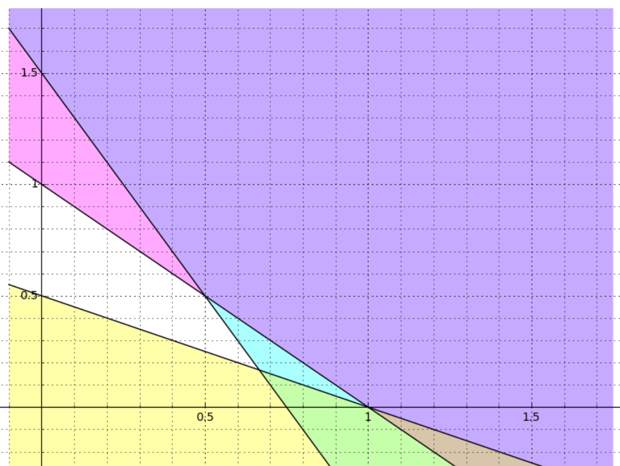
$$\begin{aligned} y &= 1.5 - 2x \\ y &= 1 - x \\ y &= 0.5 - 0.5x \\ x &= 0 \\ y &= 0 \end{aligned}$$

and then shade the regions which violate each inequality. The reason that we do this is that we will see a white region in the center of the graph. This white region is the place where all the inequalities are simultaneously satisfied—called the "feasible region" when teaching *Operations Research*, *Industrial Engineering*, or *Linear Programming*.

The code that does this is below.

```
P1 = plot( 1.5-2*x, (x, -0.1, 1.75), ymin=-0.25, ymax=1.75,
          fill=10, fillcolor='cyan', color='black', fillalpha=1/3,
          gridlines='minor' )
P2 = plot( 1-x, (x, -0.1, 1.75), ymin=-0.25, ymax=1.75,
          fill=10, fillcolor='magenta', color='black', fillalpha=1/3 )
P3 = plot( 0.5-0.5*x, (x, -0.1, 1.75), ymin=-0.25, ymax=1.75,
          fill=-10, fillcolor='yellow', color='black', fillalpha=1/3 )

P = P1 + P2 + P3

P.show()
```

This is the image produced, which is good but not entirely right. We will correct the flaw in Page 1010. **I challenge you to identify the flaw now, and I will reveal it on that page.**



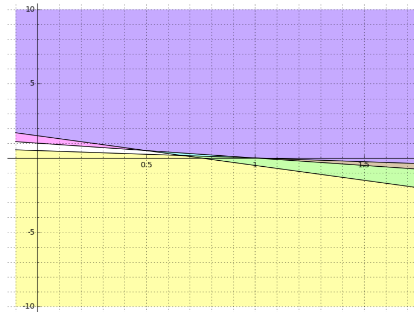
Note: Shaded points violate one or more inequalities.
Unshaded points satisfy all inequalities.

Recall, we are shading the regions of the xy-plane which violate the inequality. If instead we were to shade the "satisfied regions" which do not violate the inequality, then finding the feasible region would be very hard for a human. We'd have to search out the single region that is shaded in three different ways, distinguishing it from regions that had been shaded once or twice. This is why we always shade the "angry regions" instead of the "satisfied regions"—so that the human eye need only look for the region which is not shaded at all.

Summary of the Strategies. What has changed here, compared to graphing a single inequality? First, we are adding three plots, which means that Sage will superimpose these plots. That's how we produce the different types of shading. Second, we've reduced the `fillalpha` parameter to accommodate overlaying—otherwise the regions with two or three overlapping

colors will look too dark to the human eye. Third, we are using a different `fillcolor` choice for each inequality. This will help the reader understand which inequality or inequalities is causing a particular region to be declared infeasible. It is important to choose pastels, otherwise the resulting image is too dark to be readable. Fourth, I added `color='black'` to help make the boundaries of each inequality stand out more to the human eye. Fifth, we only need the `gridlines` optional parameter in the first plot, which will be on the bottom of the super-imposition.

As before, we really do have to restrict the x-coordinates and the y-coordinates. Otherwise, each of the three plots individually will have a different upper-bound and lower-bound for y, which makes the resulting overlay look very silly and incorrect. Below is the graph that you get if you remove the `ymin` and `ymax` optional parameters. In that graph, the upper-bound is $y = 10$ and the lower-bound is $y = -10$, because of how we did the filling.



Note: This is what happens if you remove the `ymin` and `ymax` from each plot.

Second Attempt

The flaw in our first attempt has to do with the fact that in Operations Research, Industrial Engineering, or Linear Programming, we (almost always) restrict our variables to be positive. This means that there are two "implied" inequalities, namely $x \geq 0$ and $y \geq 0$. Our lovely plot should be corrected to accommodate that. It turns out that it will look even better after we make that correction.

Just as we drew the line $y = 0.5 - 0.5x$ to plot the inequality $y \geq 0.5 - 0.5x$, we will draw the line $y = 0$ to plot the inequality $y \geq 0$. That's just a line of slope zero, that is to say, a horizontal line. Intuitively, to plot the inequality $x \geq 0$, you might think that we want to draw the line $x = 0$. The vertical line $x = 0$ is a bit problematic, because it is not in $y = mx + b$ form. The slope of the line is "undefined" in mathematics, but weak students (and some people with a PhD in physics) will say that the slope is infinity. We could use `implicit_plot` if we wanted to, but there is an easier way.

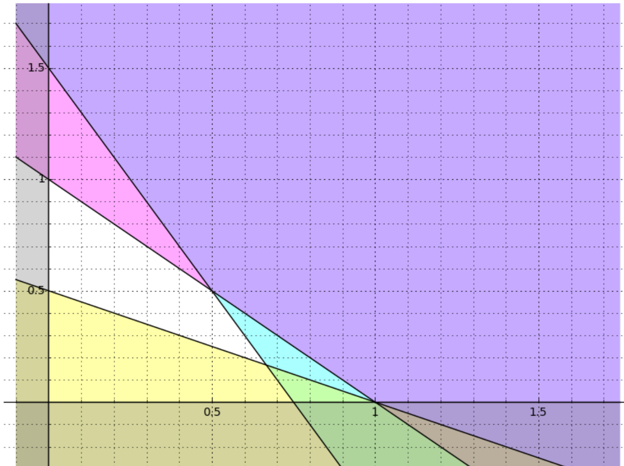
Taking inspiration from the idea of the slope being infinity, we will draw the vertical line through the origin by instead drawing the line $y = 10^9x$.

The slope being "one billion" is going to make the line appear vertical. Then we can treat it like any other line.

We now have the following code:

```
P1 = plot( 1.5-2*x, (x, -0.1, 1.75), ymin=-0.25, ymax=1.75,
          fill=10, fillcolor='cyan', color='black', fillalpha=1/3,
          gridlines='minor' )
P2 = plot( 1-x, (x, -0.1, 1.75), ymin=-0.25, ymax=1.75,
          fill=10, fillcolor='magenta', color='black', fillalpha=1/3 )
P3 = plot( 0.5-0.5*x, (x, -0.1, 1.75), ymin=-0.25, ymax=1.75,
          fill=-10, fillcolor='yellow', color='black', fillalpha=1/3 )
P4 = plot( 0, (x, -0.1, 1.75), ymin=-0.25, ymax=1.75,
          fill=-10, fillcolor='gray', color='black', fillalpha=1/3 )
P5 = plot( (10^9)*x, (x, -0.1, 1.75), ymin=-0.25, ymax=1.75,
          fill=10, fillcolor='gray', color='black', fillalpha=1/3 )

P = P1 + P2 + P3 + P4 + P5

P.show()
```

The code above produces the beautiful image below. Observe that I use gray for both of the "trivial inequalities" $x \geq 0$ and $y \geq 0$ to show that, even collectively, they are constraints of a non-primary nature.



Note: Shaded points violate one or more inequalities.
Unshaded points satisfy all inequalities.

The white region in the middle of the graph is the feasible region of the system of inequalities.

A Challenge for You:

At this point, try to make a plot of the following system of linear inequalities:

$$\begin{aligned} 3x+5y &\geq 15 \\ 500x+300y &\leq 1500 \\ 40x+40y &\leq 160 \\ x &\geq 0 \\ y &\geq 0 \end{aligned}$$

Theoretical Ramifications

The next three paragraphs contain some interesting facts which apply to any system of linear inequalities in two variables.

There is a cool theorem that says the feasible region will always be a convex polygon, so long as the feasible region is bounded. In this context, bounded means that there exists some circle (regardless of where it is centered and of what radius) which entirely contains the whole feasible region.

Another cool theorem about two-variable systems of linear inequalities, sometimes called "The Fundamental Theorem of Linear Programming," states that any linear "objective function," such as $3x+5y$, $200x-500y$, or x, will achieve a global feasible minimum and a global feasible maximum, provided that the feasible region is bounded. While there can be infinitely many global feasible minimums (or maximums) on rare occasions, there is usually a unique global feasible maximum and a unique global feasible minimum. Moreover, even when there are infinitely many, at least two of those global feasible minimums (or maximums) are guaranteed to be located on distinct corners of the feasible region. Finding such points is the heart of linear programming and the purpose of "The Simplex Method." That algorithm, its generalizations, and its successors, are the driving force for much of Operations Research and Industrial Engineering, though usually using hundreds or thousands of variables, not just two variables.

In fact, even if the feasible region is unbounded (and therefore not a polygon) it will be an intersection of some finite number of half-planes. Moreover, either the global feasible maximum will not exist (meaning that the objective function can be made arbitrarily large while still restricting to feasible points) or global feasible maxima will exist. The former case, where the objective function can be made arbitrarily large, usually represents some sort of mistake on the part of the human writing down the system of inequalities, because we cannot honestly expect to have infinite profit, infinite revenue, or to produce an infinite amount of some product. In the latter case, where global feasible maxima exist, either there is a unique global feasible maximum and it is a corner point of the feasible region, or alternatively, infinitely many exist and at least two of them are guaranteed to be distinct corners of the feasible region—exactly the situation when the

feasible region was bounded. The analogous statements are also true for global feasible minima.

G.2.3. Plotting Non-Linear Inequalities (2D Region Plots)

Normally inequalities deal with relatively straight-forward functions, such as bijections, which can be handled easily with the above techniques. Once in a while, a mathematician might be curious about a non-invertible non-linear[1] function used in an inequality.

The region_plot command deals with plotting the set of points (x, y) which satisfy a two-variable inequality in x and y. Often, the relationship between x and y cannot be represented as a function.

For example, to plot the set of points (x, y) such that

$$\cos\left(x^2 + y^2\right) \leq 0$$

for $-5 \leq x \leq 5$ and $-5 \leq y \leq 5$, we would type

```
var("y")
region_plot(cos(x^2+y^2) <= 0, (x, -5, 5), (y, -5, 5) )
```

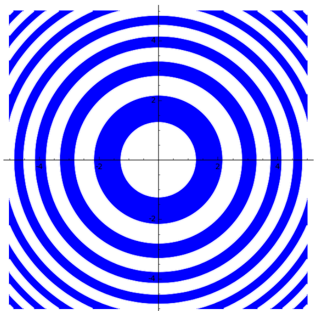
Similarly, to plot the set of points (x, y) such that

$$x^2 + 4(y^3 - y) \leq 10$$

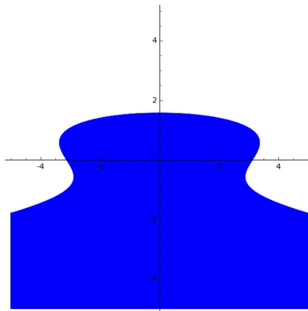
for $-5 \leq x \leq 5$ and $-5 \leq y \leq 5$, we would type

```
var("y")
region_plot( x^2+4*(y^3-y) <= 10, (x, -5, 5), (y, -5, 5) )
```

The plots produced by those two commands are



$\cos\left(x^2 + y^2\right) \leq 0$



$x^2 + 4(y^3 - y) \leq 10$

By the way, region_plot has several options, and you can read about them by typing region_plot? and reading the online help. Those options can give the regions colors and borders (dashed, dotted, or solid).

[1]The examples in this section were recommended in the "PREP Tutorial: Advanced 2D plotting." That tutorial was developed during the Mathematical Association of America PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by the National Science Foundation Department of Undergraduate Education grant # 0817071).

A Challenge for You:

Trust me—this one is really worth doing. A fun inequality is the subset of the xy-plane where

$$(\sin x)(\sin y) > 1/4$$

restricting to perhaps $-10 < x < 10$ and $-10 < y < 10$.

G.3. Heat Maps, Contour Plots, and Density Plots

We learned about Contour Plots in Section 3.5 of *Sage for Undergraduates.* This section will greatly expand upon those capabilities, including some beautiful color images. You might want to refresh your memory by rereading Section 3.5 before continuing onward.

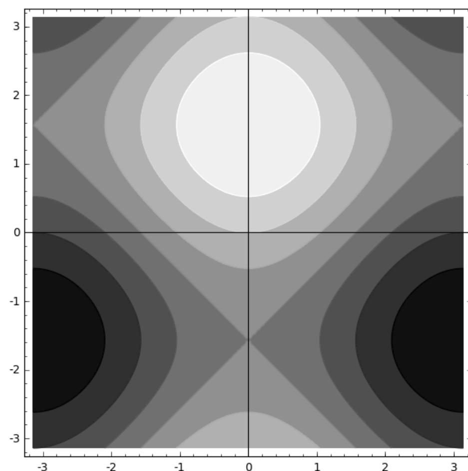
G.3.1. Contour Plots vs Density Plots in Black-and-White

I'd like to now introduce you to a distinction between contour maps and density plots. They're almost the same thing, so this can be better explained by example.

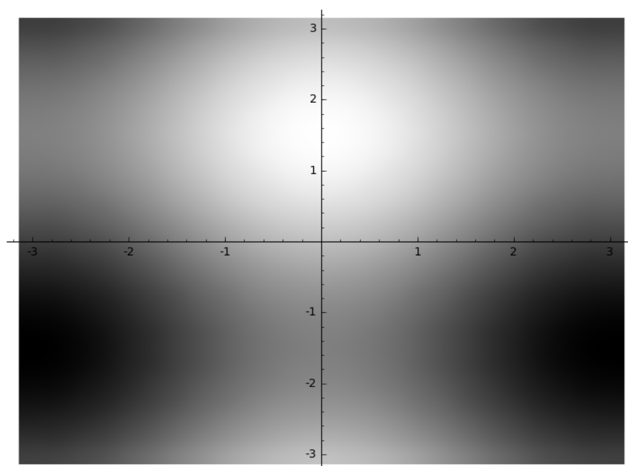
The code:

```
f(x,y) = cos(x) + sin(y)
contour_plot( f(x,y), (x,-pi,pi), (y,-pi,pi), axes=True)
```

creates this image:



Contrastingly, just changing the `contour_plot` into `density_plot`, results in this image:

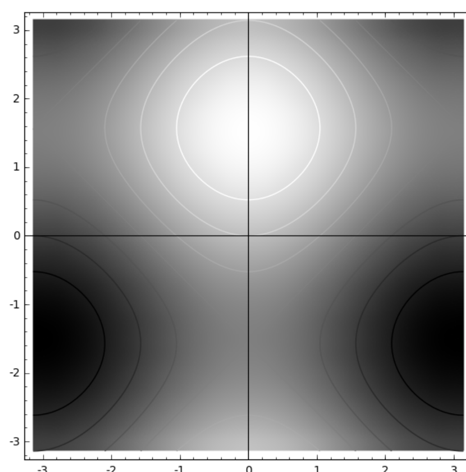


As you can see, with the contour plot the z-coordinate takes the values of $f(x, y)$ and is divided into "levels" that are discrete. That results in sharp edges between boundaries. With the density plot, the z-coordinate is allowed to vary continuously, and you get a blend of shades with no sharp edges.

It is also possible to combine the two outputs, to make the clear boundaries significantly more visible. The following code

```
f(x,y) = cos(x) + sin(y)
P1=contour_plot(f(x,y), (x,-pi,pi), (y,-pi,pi), axes=True, fill=false)
P2=density_plot(f(x,y), (x,-pi,pi), (y,-pi,pi), axes=True)

show(P1+P2)
```

creates this image:



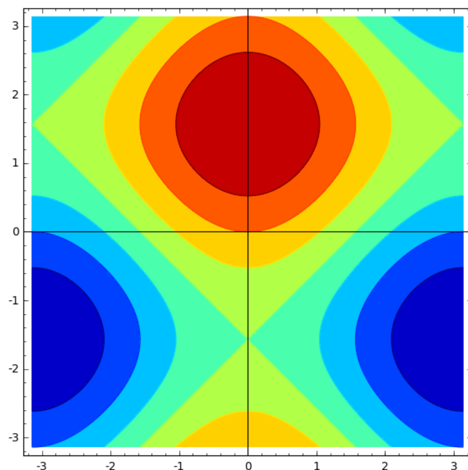
As you can see, these plots are not very impressive when done in black-and-white. In the next section we will plot them in color, and the visual appeal will be much improved.

G.3.2. Color Density Plots and Contour Plots

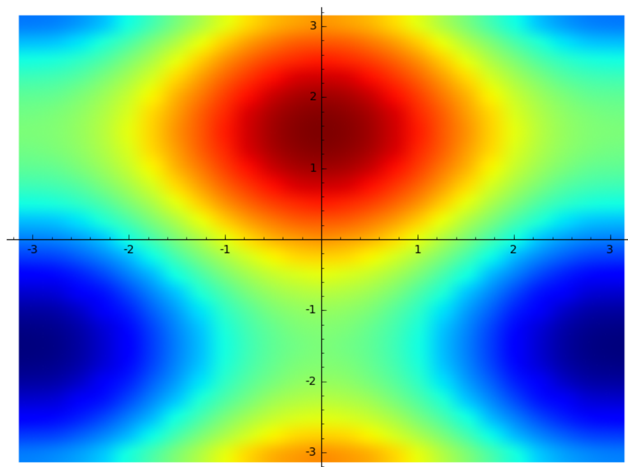
The plots in the previous subsection will become much more visually appealing if drawn in full color. We will now re-plot those images. The code

```
f(x,y) = cos(x) + sin(y)
contour_plot( f(x,y), (x,-pi,pi), (y,-pi,pi), axes=True, cmap='jet')
```

creates this image:



The only change was to add the option `cmap='jet'` to the `contour_plot` command. Similarly, changing `contour_plot` into `density_plot` results in this image:



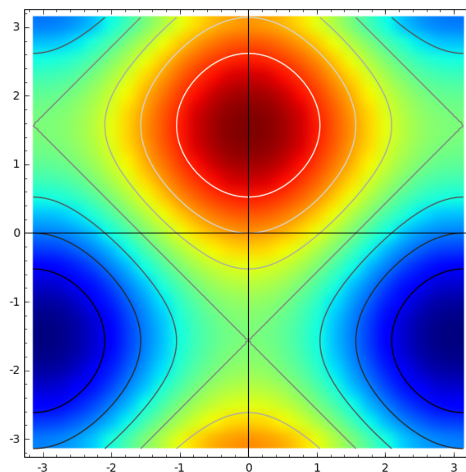
which varies continuously. By the way, an image of this type is sometimes called a "heat map" in the popular culture of Data Science. A heat map is just a color density map of a two-variable function.

We can also make the combined plot drawn in color. The following code

```
f(x,y) = cos(x) + sin(y)
P1=contour_plot(f(x,y), (x,-pi,pi), (y,-pi,pi), axes=True, fill=false)
P2=density_plot(f(x,y), (x,-pi,pi), (y,-pi,pi), axes=True, cmap='jet')
```

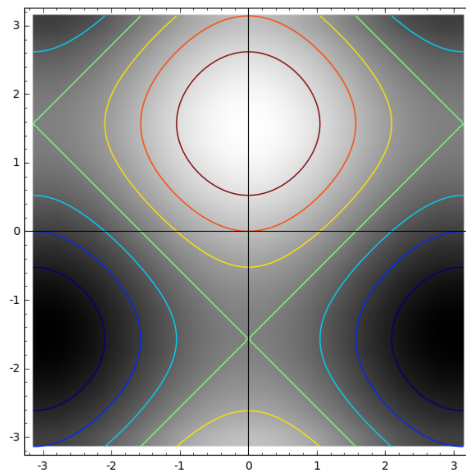
```
show(P1+P2)
```

results in this very clear image:



It is important to note that I added `cmap='jet'` only to the density plot, and not to the contour plot. This was to force the contour plot to produce black-and-white boundaries. If you add `cmap='jet'` to both commands, then you'll see that the color boundaries between regions are almost invisible.

Alternatively, one can add `cmap='jet'` only to the contour plot but not to the density plot. That will produce the following image, which looks cool but which is not necessarily mathematically informative.



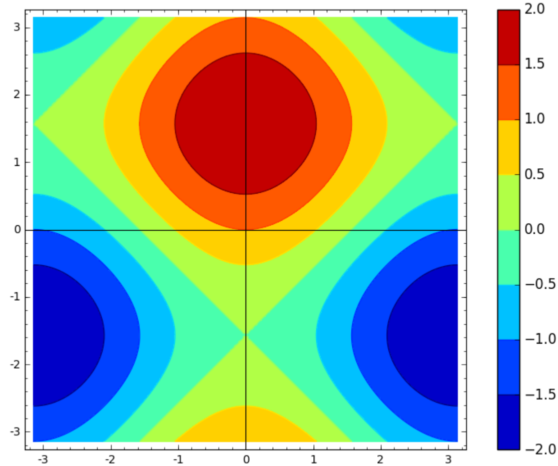
G.3.3. Adding a Legend or Color Bar

With contour plots it can be very useful to add a legend, so that the reader knows what the colors actually represent. The following code

```
f(x,y) = cos(x) + sin(y)
contour_plot( f(x,y), (x,-pi,pi), (y,-pi,pi), axes=True,
```

```
    cmap='jet', colorbar=True)
```

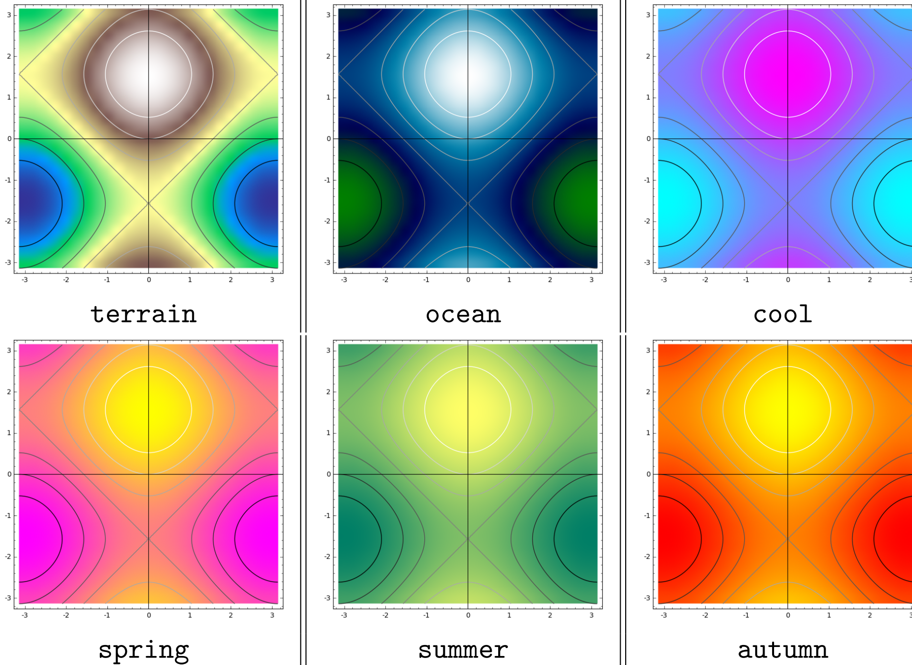
produces this image:



G.3.4. Other Color Maps

There are a lot of built-in color maps (the `cmaps`). We've seen `jet`, which is the most common. There is also `gray`, which is the default used when you don't use the cmap parameter at all. A complete list is given in Table 1 on Page 1019.

The only way to get to know the color maps (the `cmaps`) is to experiment and see which ones match well with the data that you're trying to present. Here are six interesting examples to whet your appetite.



Spectral	summer	coolwarm	Wistia_r	pink_r	Set1
Set2	Set3	brg_r	Dark2	hot	PuOr_r
afmhot_r	terrain_r	PuBuGn_r	RdPu	gist_ncar_r	gist_yarg_r
Dark2_r	YlGnBu	RdYlBu	hot_r	gist_rainbow_r	gist_stern
gnuplot_r	cool_r	cool	gray	copper_r	Greens_r
GnBu	gist_ncar	spring_r	gist_rainbow	RdYlBu_r	gist_heat_r
Wistia	OrRd_r	CMRmap	bone	gist_stern_r	RdYlGn
Pastel2_r	spring	terrain	YlOrRd_r	Set2_r	winter_r
PuBu	RdGy_r	spectral	flag_r	jet_r	RdPu_r
Purples_r	gist_yarg	BuGn	Paired_r	hsv_r	bwr
cubehelix	YlOrRd	Greens	PRGn	gist_heat	spectral_r
Paired	hsv	Oranges_r	prism_r	Pastel2	Pastel1_r
Pastel1	gray_r	PuRd_r	Spectral_r	gnuplot2_r	BuPu
YlGnBu_r	copper	gist_earth_r	Set3_r	OrRd	PuBu_r
ocean_r	brg	gnuplot2	jet	bone_r	gist_earth
Oranges	RdYlGn_r	PiYG	CMRmap_r	YlGn	binary_r
gist_gray_r	Accent	BuPu_r	gist_gray	flag	seismic_r
RdBu_r	BrBG	Reds	BuGn_r	summer_r	GnBu_r
BrBG_r	Reds_r	RdGy	PuRd	Accent_r	Blues
Greys	autumn	cubehelix_r	nipy_spectral_r	PRGn_r	Greys_r
pink	binary	winter	gnuplot	RdBu	prism
YlOrBr	coolwarm_r	rainbow_r	rainbow	PiYG_r	YlGn_r
Blues_r	YlOrBr_r	seismic	Purples	bwr_r	autumn_r
ocean	Set1_r	PuOr	PuBuGn	nipy_spectral	afmhot

Table 1. A listing of all the color maps built into Sage (as of July 2016).

By the way, the ending `_r` means "reversed." For example, the color map `terrain_r` is actually the color map `terrain` but using $1 - z$ in place of z.

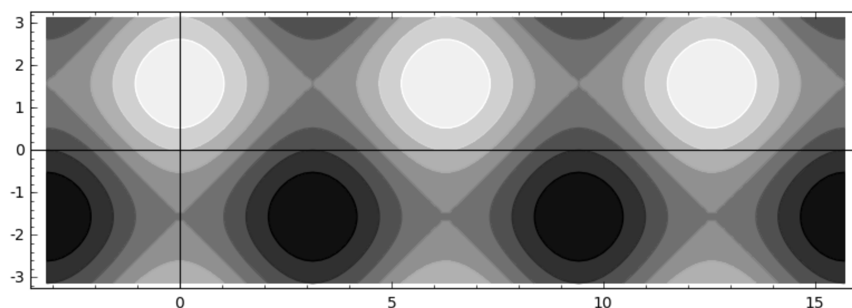
By the way, we can also use color maps for 3D-functions. See Section H.1.4.

G.3.5. Aspect Ratios in Contour Plots and Density Plots

We did not specify any aspect ratios in any of the code so far, throughout Section G.3. That's because the two commands `contour_plot` and `density_plot` will compute an aspect ratio for you. The problem is that they do it very differently. On the one hand, the code

```
f(x,y) = cos(x) + sin(y)
contour_plot( f(x,y), (x,-pi,5*pi), (y,-pi,pi), axes=True)
```

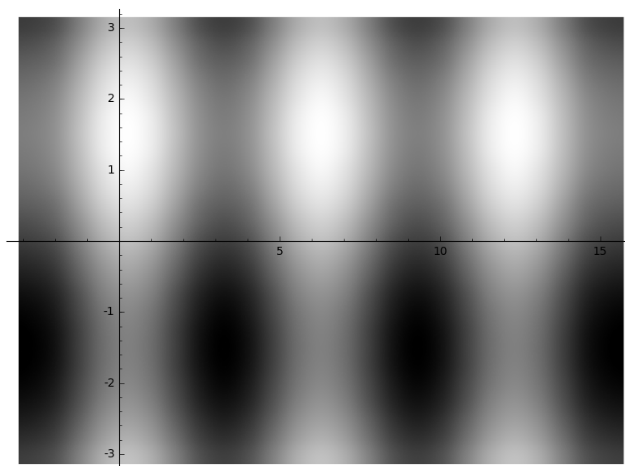
creates this image:



On the other hand, the code

```
f(x,y) = cos(x) + sin(y)
density_plot( f(x,y), (x,-pi,5*pi), (y,-pi,pi), axes=True)
```

creates this image:

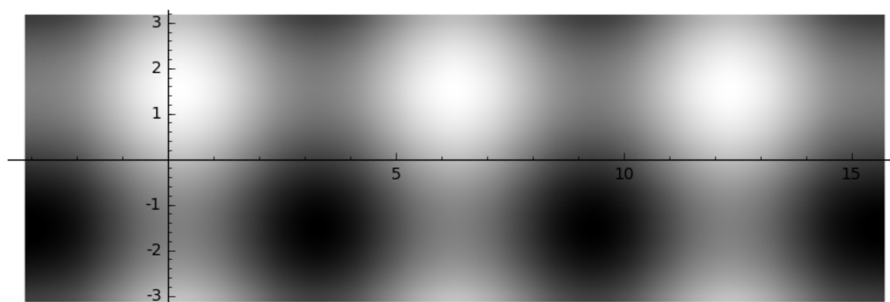


Some would argue that the first image is much better. That's because the level set associated with the white region is actually a circle, mathematically. Therefore, it should appear as a circle on the computer screen. However, that image has unusual dimensions, in that it is much wider than it is tall. The second image has much more common dimensions, yet the region associated with the level set is now a blurry white ellipse, instead of a blurry white circle.

The "correct approach" depends on the application. In any case, you can always force Sage to use whatever aspect ratio you would like. For example, to force that density plot above to use circles, type the code

```
f(x,y) = cos(x) + sin(y)
density_plot( f(x,y), (x,-pi,5*pi), (y,-pi,pi), axes=True,
    aspect_ratio=1)
```

It makes the image below, where the white region is now a blurry circle.



G.3.6. An Example from Mathematical Economics

Let's say that there's a company that sells gadgets. The gadget comes in two models: the deluxe model and the regular model. At this time, the demand is so high that the products are being sold immediately and are on back-order. Production can be increased to about four times current levels. Naturally, management wants to raise the prices, to take advantage of the high demand. The goal is to compute the optimal price scheme.

Here is the given information:

- The deluxe gadget currently sells for 79 dollars each, and has a demand of 25,000 units per year.
- The regular gadget currently sells for 49 dollars each, and has a demand of 55,000 units per year.
- The marketing department indicates that each dollar increase of the deluxe gadget will reduce the demand of the deluxe gadget by 1100 and increase the demand of the regular gadget by 800.
- The marketing department also indicates that each dollar increase of the regular gadget will reduce the demand of the regular gadget by 1900 but increase the demand of the deluxe gadget by 1400.
- The deluxe gadget costs 25 dollars to manufacture, and the regular gadget costs 20 dollars to manufacture.

First, we start by computing the demand function of each gadget.

- Let x be the price of the deluxe gadget, and y be the price of the regular gadget.
- The demand of the regular gadget is

$$f_R(x, y) = 55,000 - 1900(y - 49) + 800(x - 79)$$

  because $(y-49)$ is the number of dollars that the regular price goes up by, and $(x - 79)$ is the number of dollars that the deluxe gadget goes up by.
- The demand of the deluxe gadget is

$$f_D(x, y) = 25,000 + 1400(y - 49) - 1100(x - 79)$$

  for similar reasons.

- The revenue function can be computed by

$$\begin{aligned} R(x,y) &= xf_D(x,y) + yf_R(x,y) \\ &= x\,(25,000 + 1400(y-49) - 1100(x-79)) + \\ &\quad y\,(55,000 - 1900(y-49) + 800(x-79)) \end{aligned}$$

- Note, there is no need to simplify that, since a computer will do most of the work.
- The cost function can be computed by

$$\begin{aligned} C(x,y) &= 25f_D(x,y) + 20f_R(x,y) \\ &= 25\,(25,000 + 1400(y-49) - 1100(x-79)) + \\ &\quad 20\,(55,000 - 1900(y-49) + 800(x-79)) \end{aligned}$$

- The profit function can be computed by

$$\begin{aligned} P(x,y) &= R(x,y) - C(x,y) \\ &= x\,(25,000 + 1400(y-49) - 1100(x-79)) + \\ &\quad y\,(55,000 - 1900(y-49) + 800(x-79)) - \\ &\quad 25\,(25,000 + 1400(y-49) - 1100(x-79)) - \\ &\quad 20\,(55,000 - 1900(y-49) + 800(x-79)) \end{aligned}$$

- The profit function could also have been computed by

$$\begin{aligned} P(x,y) &= (x-25)f_D(x,y) + (y-20)f_R(x,y) \\ &= (x-25)\,(25,000 + 1400(y-49) - 1100(x-79)) + \\ &\quad (y-20)\,(55,000 - 1900(y-49) + 800(x-79)) \end{aligned}$$

  because each deluxe model yields a profit of $(x-25)$ dollars and each regular model yields a profit of $(y-20)$ dollars.
- You might want to take a moment to verify that the two profit functions above are actually the same function.
- To be honest, I should note that a more common notation in microeconomics would be to write $D_X(x,y)$ and $D_Y(x,y)$, in place of $f_R(x,y)$ and $f_D(x,y)$, respectively. However, in mathematics D signifies a derivative. Moreover, $D_X(x,y)$ looks alarmingly similar to Einstein notation for partial derivatives in the theory of relativity, where $D_x(x,y) = \partial D(x,y)/\partial x$. I find Einstein notation to be rather nice, because f_{xxxyy} looks much simpler than $\partial^5 f/\partial x^3 \partial y^2$. Let us now return to the problem at hand.

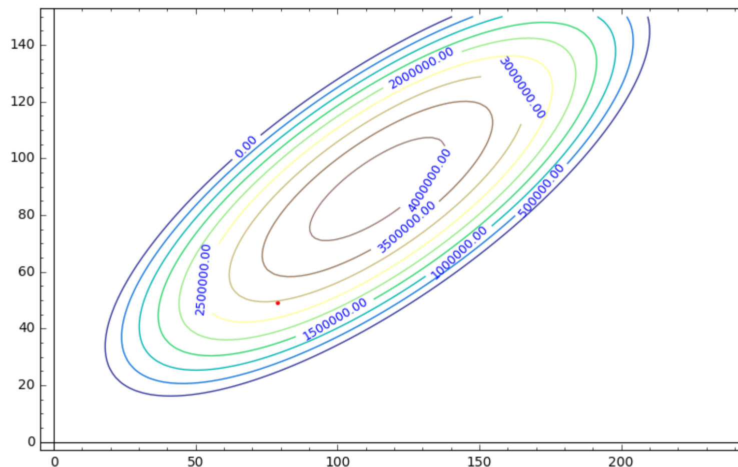
As a sanity check, we should verify the current profits. On a hand calculator, we can compute

$$\underbrace{(79)(25000) + (49)(55000)}_{\text{revenue}} - \underbrace{(25)(25000) - (20)(55000)}_{\text{costs}} = \underbrace{2,945,000}_{\text{profit}}$$

This is the code that I wrote to investigate this problem.

```
var("x y")

M = 1000*1000

P(x,y) = (x-25)*(25000+1400*(y-49)-1100*(x-79)) +
             (y-20)*(55000-1900*(y-49)+800*(x-79))

print "Sanity Check: ", P(79, 49)

contour_plot( P(x,y), (x,0,240), (y,0,150),
         contours = [0*M, 0.5*M, 1*M, 1.5*M, 2*M, 2.5*M, 3*M, 3.5*M,
         4*M, 4.5*M, 5*M], axes=True, fill=false, labels=true,
         cmap='terrain', label_inline=true ) + point( (79,49),
         color='red' )
```

You'll notice that I defined `M` to be one million dollars. Then the contours represent iso-profit curves for break-even up to five million, with half-million increments. I considered a range of prices from zero up to triple the current prices. Here is the image produced by that code.



The red dot represents the current price point, which is far from optimal. You will also note that there is no curve for 5,000,000 nor 4,500,000. Those profit levels are not achievable. Now that we have a nice plot, and that we have computed these functions, a simple calculus computation will reveal the optimal prices.

$$
\begin{aligned}
P(x,y) &= (x-25)(25000+1400(y-49)-1100(x-79))+ \\
&\quad (y-20)(55000-1900(y-49)+800(x-79)) \\
\partial P(x,y)/\partial x &= (1)(25000+1400(y-49)-1100(x-79))+(x-25)(0+0-1100)+ \\
&\quad (0)(55000-1900(y-49)+800(x-79))+(y-20)(0-0+800) \\
&= 25000+1400(y-49)-1100(x-79)+(x-25)(-1100)+(y-20)(800) \\
&= 25000+1400y-68600-1100x+86900-1100x+27500+800y-16000 \\
&= 54800+2200y-2200x \\
\partial P(x,y)/\partial y &= (0)(25000+1400(y-49)-1100(x-79))+(x-25)(0+1400-0)+ \\
&\quad (1)(55000-1900(y-49)+800(x-79))+(y-20)(0-1900+0) \\
&= 1400(x-25)+55000-1900(y-49)+800(x-79)-1900(y-20) \\
&= 1400x-35000+55000-1900y+93100+800x-63200-1900y+38000 \\
&= 2200x+87900-3800y
\end{aligned}
$$

Sage can do all this work for me, automatically, with the following commands:

```
P(x,y) = (x-25)*(25000+1400*(y-49)-1100*(x-79))
          + (y-20)*(55000-1900*(y-49)+800*(x-79))

P(x,y).gradient()
```

which give the same answer, but much more quickly. However, I didn't realize that until after I had done the above work by hand. The output from Sage is as follows:

```
(-2200*x + 2200*y + 54800, 2200*x - 3800*y + 87900)
```

We can solve for the optimal price now, with the following code:

```
var("x y")
solve( [-2200*x + 2200*y + 54800 == 0,
          2200*x - 3800*y + 87900 == 0], [x,y] )
```

We obtain $x = 20081/176 = 114.096\cdots$ and $y = 1427/16 = 89.1875$ giving us the price points of \$ 114.10 for the deluxe model, and \$ 89.19 for the regular model. The total profit is \$ $4,265,537.21\cdots$, though obviously nine digits of precision is totally unwarranted.

Note that we did not maximize revenue, nor did we maximize sales. We certainly did not minimize costs. These are four separate objectives: maximizing revenue, minimizing cost, maximizing profit, and maximizing sales.

- Our plan calls for the deluxe model having a price of \$ 114.10 and the regular model having a price of \$ 89.19. That would result in 6719 regular units and 42,656 deluxe units being sold. The revenue

would be $ 5,466,317, the costs would be $ 1,200,780, and a profit of $ 4,265,537.

- It is a good exercise for the reader to optimize $R(x, y)$ instead of $P(x, y)$.
- Maximizing revenue would have a price point of $ 99.81 for the deluxe model and $ 80.13 for the regular model. This results in selling 45,691 deluxe models and 12,501 regular models. The revenue is $ 5,562,123 but the costs are $ 1,392,295, for a profit of $ 4,169,828. As you can see, the profit is inferior to our optimization of profit. However, the revenue is superior.
- However, both plans are better than the original plan. The original sold 55,000 regular gadgets at $ 49 each, and 25,000 gadgets at $ 79 each. This results in a revenue of $ 4,670,000 but a cost of $ 1,725,000, thus a profit of $ 2,495,000.
- Minimizing costs would require producing nothing, which is not a good plan for a business.
- The production limits were roughly $4(55,000) = 220,000$ regular gadgets and roughly $4(25,000) = 100,000$ deluxe gadgets. As you can see, that will not impact our plans.

The following code is very useful for experimentation:

```
var("x y")

f_R(x,y) = (55000-1900*(y-49)+800*(x-79))
f_D(x,y) = (25000+1400*(y-49)-1100*(x-79))
R(x,y) = (x)*(25000+1400*(y-49)-1100*(x-79))
            + (y)*(55000-1900*(y-49)+800*(x-79))
C(x,y) = 25*(25000+1400*(y-49)-1100*(x-79))
            + 20*(55000-1900*(y-49)+800*(x-79))
P(x,y) = (x-25)*(25000+1400*(y-49)-1100*(x-79))
            + (y-20)*(55000-1900*(y-49)+800*(x-79))

sample_x = 79
sample_y = 49

print "Demand (regular): ", f_R(sample_x, sample_y)
print "Demand (deluxe): ", f_D(sample_x, sample_y)
print "Revenue: ", R(sample_x, sample_y)
print "Cost: ", C(sample_x, sample_y)
print "Profit: ", P(sample_x, sample_y)
```

A Challenge for You:

To confirm that you understood the above work, you could challenge yourself to produce the correct contour plot for the revenue function. You should also challenge yourself to come up with the optimal x and y for maximizing

revenue, instead of profit. Those x and y values are given in the bullet list above, so you will know whether you've done it correctly or not.

As a further challenge, suppose that the cost of microelectronics is increased slightly due to currency fluctuations. The manufacturing cost of the deluxe unit rises from 25 to 27 dollars, and the manufacturing cost of the regular unit rises from 20 to 21 dollars. Recompute the price points that optimize profit as well as the price points that optimize revenue in light of these increased costs.

G.3.7. Dangerous Examples: The Pitfalls of Density Plots

I'd like to share two examples with you now, that show how density plots and heat maps can be misused.

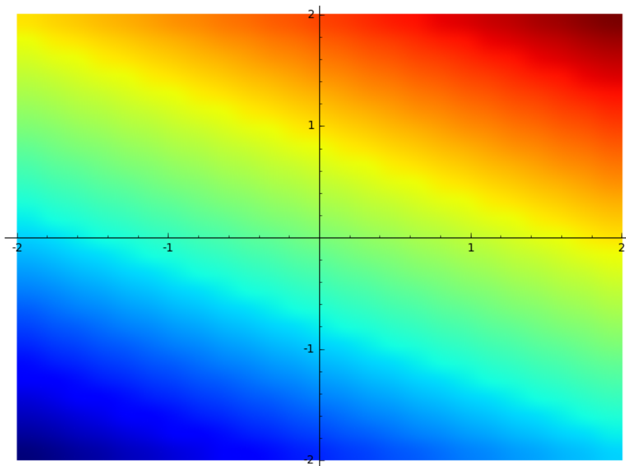
Example One: Over-exaggerating Features

While this example might seem a bit contrived, it relates to something that I've actually seen happen. I'll give the full story after giving the Sage example.

Consider the following function $f(x, y)$, which describes the temperature of a 4×4 metal plate, using the coordinates $-2 < x < 2$ and $-2 < y < 2$.

$$f(x, y) = 3500 + x + 2y$$

Clearly, the temperature doesn't change very much at all. The temperature ranges from 3494 to 3506 degrees. The plate can be said to be of almost constant temperature. Here is a color density plot of the function—a heat map.



As you can see, the graphic makes it look as though the temperature changes tremendously across the surface of the plate. By the way, the code for that image is below.

```
f(x,y) = 3500 + x + 2*y
```

```
density_plot(f, (x,-2,2), (y,-2,2), cmap='jet' )
```

One of my professors (I have forgotten who, unfortunately) had assigned an interesting computer project: a simulation to describe the cooling of a metal plate. This involves some rudimentary PDEs (partial differential equations) and a knowledge of thermodynamics, as well as some relatively basic computer programming—so it was a good project.

A graduate student of his had the code display a heat map (a color density plot) to display the final solution. In this case, the final solution should have been a plate of constant temperature. That's a boring image, but it would help verify that the program was converging to the correct final answer. The student was expecting a large rectangle of a constant color. Instead, the student got a final image that looked like what we saw above. The student was very dismayed, and lost a lot of time trying to figure out what was wrong with the code.

After much frustration and many days of re-examining his code, he discovered that absolutely nothing was wrong at all. The code was working perfectly. The variation being displayed in the image was rounding error, on the order of 10^{-12}. This error was there because of the fact that the floating-point arithmetic used by the computer does not have infinite precision.

Of course, all of this stress could have been avoided if the student had followed the advice that I gave in Section G.3.3, "Adding a Legend or Color Bar."

Example Two: Under-exaggerating Features

Let's suppose that one is investigating[2] the function:

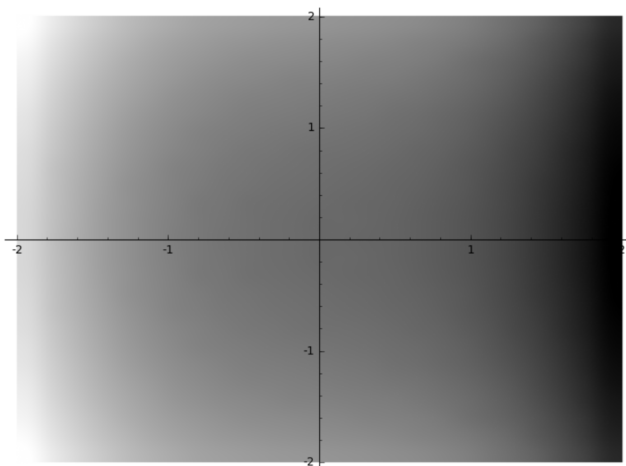
$$f(x, y) = -x^3 + y^2 - x + 1$$

A simple, unadorned density plot would give the impression that the function doesn't change very much, and that it has very little structure. One way to plot that function, without first thinking about it mathematically, would be to type the following code:

```
f(x,y) = -x^3 + y^2 - x + 1
density_plot(f, (x,-2,2), (y,-2,2) )
```

They would get the following image, which would deceive them. It would imply that the function doesn't vary significantly. Even worse, it reveals no structure at all.

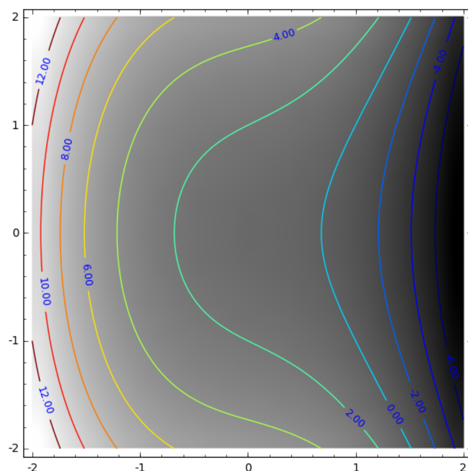
[2]This brilliant example was recommended in the "PREP Tutorial: Advanced 2D plotting." That tutorial was developed during the Mathematical Association of America PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by the National Science Foundation Department of Undergraduate Education grant # 0817071).



A more careful and thoughtful image can be produced by the following code:

```
f(x,y) = -x^3 + y^2 - x + 1
P1 = density_plot(f, (x,-2,2), (y,-2,2) )
P2 = contour_plot(f, (x,-2,2),(y,-2,2), fill=False, labels=True,
          label_inline=True, cmap='jet',
          contours = [-6,-4,-2,0,2,4,6,8,10,12] )

show(P1+P2)
```

As you can see below, the function has a great deal of structure. The level sets form an ensemble of elliptic curves—curves that are very important in cryptography.



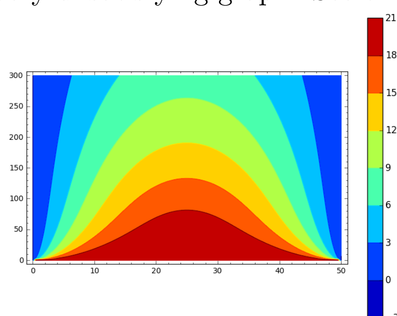
Example Three: Forgotten Aspect Ratios

When making contour plots, things can go very wrong if you do not compute an aspect ratio. In Section 3.5.1 of *Sage for Undergraduates*, we saw an

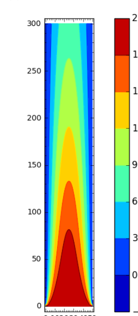
example from thermodynamics where we had (almost) the following code. The only addition is that I've added `cmap='jet'` to add color to the graph.

```
var("j")

u(x, t) = (80/pi)*sum( (1/(2*j+1))*exp(-(2*j+1)^2*pi^2*t/2500.0)
    * sin((2*j+1)*pi*x/50), j, 0, 100)

contour_plot( u(x,t), (x,0,50), (t,0,300), colorbar=True, aspect_ratio=0.1,
    cmap='jet' )
```

If you backspace over the `aspect_ratio = 0.1` option, then you'll get a very unsatisfying graph. See the comparison below.



With `aspect_ratio = 0.1` Option
(i.e. aspect ratio set to 0.1)



Without `aspect_ratio = 0.1` Option
(i.e. default aspect ratio of 1.0)

G.4. Saving your Image Files

There are numerous useful ways to save your images to a file in Sage, and mercifully they all use exactly the same syntax. You can pick from many file formats. The extension of the filename that you specify will allow Sage to identify what format you want.

- The format `*.png` is great for documents on the web, or for insertion into Microsoft Word documents. This format was once rare, but is now relatively common.
- The format `*.pdf` is one that you clearly are familiar with, since you are reading a `*.pdf` file right now.
- The format `*.pgf` is similar to `*.jpg`, in the sense that it uses compression methods. This format is somewhat rare at this time (December of 2016).

The above formats store images in a *raster-graphics format*. That means that they start with a bitmap (a pixel-by-pixel storage of an image), and then do a long and complex sequence of mathematical operations to compress the image. A more powerful way of storing images is the *vector-graphics format*. This process does not start with a bitmap. Instead, it starts with a geometric definition of your image. Therefore, you can zoom in infinitely far, and still

have a clear and crisp image. With raster-graphics, once you zoom in very far, the image looks "blocky" because the pixels have become large. The following extensions are for vector-graphics formats.

- The formats `*.ps` and `*.eps` are great for printing professionally, such as in a textbook, a journal article, or a poster. The extensions stand for "postscript," a computer language mostly spoken by printers and plotters, and "encapsulated postscript."
- The format `*.svg` is used by expensive graphics software, such as Adobe Illustrator and competing products.
- Last but not least, the `*.sobj` file format produces a Sage object. Such a Sage object can be loaded again later.
- If you do not specify any extension, then the empty extension will be treated the same as the `*.sobj` extension.

The `save` feature has the following syntax:

```
P = plot( sin(x), -20, 20 )
P.show()
P.save("test1.png")
```

If you're using SageMathCloud, the file will be deposited in the same directory as the `*.sagews` worksheet which you are working from. You do not have the chance to say where the file goes. Moreover, there is no notification that the file was successfully saved. Instead, the file is created very silently. To see it, just click on the the word "Files" with the folder icon. While the file is saved "in the cloud" and not on your local computer, you can simply download the file to have a copy of it on your computer.

If you're using SageMathCell, there will be a link. It is small and rather hard to find. It is under the black window where your plot is actually shown. It is at the lower-left hand corner of that black window. You should right-click on that link, and choose "save to file…" Then, depending on how you've configured your web browser, either you will get to choose in which directory (on your computer) the file will be saved, or the file will be automatically saved to your "Downloads" directory.

Unfortunately, when I started writing *Sage for Undergraduates*, I did not know about the true importance of using vector-graphics formats over raster-graphics formats, though I was academically aware of the technical distinction. Therefore, I used only `*.png` format images. As a result, the quality of the images in *Sage for Undergraduates* was not as good as it could have been.

G.5. The Several Uses of the `show` Command

I'm very surprised and embarrassed that I left this command out of *Sage for Undergraduates*, because it is one of my favorite commands in all of Sage. My usual use for it is when entering a large and complicated formula or function.

For example, when working with mortgages, one of the formulas that is often used is to compute PV, the "present value of a decreasing annuity." That's just the cash value of a faithful promise to deliver c dollars at m equally-spaced intervals per year, for t years, starting $(1/m)$th of a year from the present time. Using r to denote the nominal interest rate, the formula is

$$PV = c \cdot \frac{1 - (1 + \frac{r}{m})^{-mt}}{r/m}$$

To put this in plainer language, let $m = 12$ and $t = 30$, to represent the most common mortgage in the USA: a 30-year/360-month fixed-rate mortgage. The PV is the cash value of the customer's faithful promise to deliver c dollars each month, for 30 years, starting one month from the present moment.

On some morning, if I have not yet had sufficient coffee, then I'd be prone to type perhaps:

```
var("Present_Value c n r m t")

show( Present_Value == c*(1-(1+r/m)^(m*t))/r/m )

# warning: this is not correct
```

Sage displays the response:

```
13
14 var("Present_Value c n r m t")
15
16 show( Present_Value == c*(1-(1+r/m)^(m*t))/r/m )
17    (Present_Value, c, n, r, m, t)
```

$$Present_{Value} = -\frac{c\left(\left(\frac{r}{m}+1\right)^{mt}-1\right)}{mr}$$

```
18
```

That would help me realize that I had made a mistake. The mistake is `/r/m`, which should be instead `/(r/m)`. I can see the error because (in the denominator of the shown equation) I see r and m side-by-side, which is not what I had wanted. Accordingly, I make the correction as follows:

```
var("Present_Value c n r m t")

show( Present_Value == c*(1-(1+r/m)^(m*t))/(r/m) )

# note: the bug is now fixed
```

Sage now displays the response:

```
7
8  var("Present_Value c n r m t")
9
10 show( Present_Value == c*(1-(1+r/m)^(m*t))/(r/m) )
11
12    (Present_Value, c, n, r, m, t)
```

$$Present_{Value} = -\frac{cm\left(\left(\frac{r}{m}+1\right)^{mt}-1\right)}{r}$$

```
13
```

While this is not identical to what I had in mind, it is easy to see that it is mathematically equivalent. Therefore, I can have confidence that I have typed the formula correctly.

The `show` command is also very nice for matrices. Here's a quick example where you can see that the `show` command gives you a prettier output than normal, but the normal output is perfectly fine too.

```
44
45 M = matrix(QQ, 5, 5, lambda x, y: (x+1) / (y+1))
46 print M
47 show(M)
48    [  1 1/2 1/3 1/4 1/5]
      [  2   1 2/3 1/2 2/5]
      [  3 3/2   1 3/4 3/5]
      [  4   2 4/3   1 4/5]
      [  5 5/2 5/3 5/4   1]
```

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ 2 & 1 & \frac{2}{3} & \frac{1}{2} & \frac{2}{5} \\ 3 & \frac{3}{2} & 1 & \frac{3}{4} & \frac{3}{5} \\ 4 & 2 & \frac{4}{3} & 1 & \frac{4}{5} \\ 5 & \frac{5}{2} & \frac{5}{3} & \frac{5}{4} & 1 \end{pmatrix}$$

```
49
```

It is quickly worth mentioning that you can also type `latex(M)` to get the code for typesetting that matrix in the document-preparation language called LaTeX. (See also, Section 4.15 of *Sage for Undergraduates*.) Remember, it is pronounced "la-tech," like a Spanish pronunciation of the definite article "la" and "tech" being the first syllable of "technology." It *is not pronounced* like "latex," as in "latex gloves."

If you are teaching people about Cardano's cubic formula and exact algebraic roots of polynomials, then the `show` command is much better than the `print` command. As you can see by the image below, the normal output is not very readable at all, but the output typeset by `show` gets a lot closer to being readable.

```
49
50 answers = solve( x^3 + x^2 + x + 2 == 0, x)
51
52 print answers[0]
53 print answers[1]
54 print answers[2]
55
56 show( answers[0] )
57 show( answers[1] )
58 show( answers[2] )
59   x == -1/2*(1/18*sqrt(83)*sqrt(3) - 47/54)^(1/3)*(I*sqrt(3) + 1) + 1/9*(-I*sqrt(3) + 1)/(1/18*sqrt(83)*sqrt(3) - 47/54)^(1/3) - 1/3
     x == -1/2*(1/18*sqrt(83)*sqrt(3) - 47/54)^(1/3)*(-I*sqrt(3) + 1) - 1/9*(-I*sqrt(3) - 1)/(1/18*sqrt(83)*sqrt(3) - 47/54)^(1/3) - 1/3
     x == (1/18*sqrt(83)*sqrt(3) - 47/54)^(1/3) - 2/9/(1/18*sqrt(83)*sqrt(3) - 47/54)^(1/3) - 1/3
```

$$x = -\frac{1}{2}\left(\frac{1}{18}\sqrt{83}\sqrt{3} - \frac{47}{54}\right)^{\frac{1}{3}}\left(i\sqrt{3}+1\right) + \frac{-i\sqrt{3}+1}{9\left(\frac{1}{18}\sqrt{83}\sqrt{3} - \frac{47}{54}\right)^{\frac{1}{3}}} - \frac{1}{3}$$

$$x = -\frac{1}{2}\left(\frac{1}{18}\sqrt{83}\sqrt{3} - \frac{47}{54}\right)^{\frac{1}{3}}\left(-i\sqrt{3}+1\right) - \frac{-i\sqrt{3}-1}{9\left(\frac{1}{18}\sqrt{83}\sqrt{3} - \frac{47}{54}\right)^{\frac{1}{3}}} - \frac{1}{3}$$

$$x = \left(\frac{1}{18}\sqrt{83}\sqrt{3} - \frac{47}{54}\right)^{\frac{1}{3}} - \frac{2}{9\left(\frac{1}{18}\sqrt{83}\sqrt{3} - \frac{47}{54}\right)^{\frac{1}{3}}} - \frac{1}{3}$$

```
60
```

Those familiar with LaTeX will be very happy to learn that `latex(answers[2])` will give the LaTeX code for typesetting the last answer. Determining the LaTeX code for that without Sage would be very unpleasant for a human.

Last but not least, if you know what continued fractions are, you might be happy to know that `show` can display them very nicely. (You can learn more about continued fractions in Section 4.20 of *Sage for Undergraduates*.) Actually, the `show` command makes using continued fractions in Sage much more comprehensible, especially for beginners. Here is an example:

```
60
61 cf = continued_fraction( 79/141 )
62 show( cf )
63
```

$$0+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{3+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\frac{1}{5}}}}}}}$$

```
64
65
```

Again, you can get the LaTeX code for that continued fraction by typing `latex(cf)`.

Last but not least, towards the end of Section 1.4 of *Sage for Undergraduates*, we learned how to use `show` when superimposing several plots.

Appendix H

Plotting in 3D

One of my favorite features in all of Sage is the capability to plot in 3D. I find this of enormous utility in teaching classes such as *Calculus II* and *Multivariate Calculus*. There are numerous ways to plot in 3D in Sage. All of them are fairly straightforward. It is only a matter of choosing which one is right for what you have in mind.

H.1. Plotting $z = f(x, y)$ in Sage

Plotting some function like

$$z = f(x, y) = x^3 - y^4$$

is probably the single most common 3D plot. While such plots are murderously hard to draw on the board for the artistically challenged (such as myself), Sage makes excellent plots for you.

H.1.1. Brief Historical Background

When I first wrote the section for 3D plotting, before *Sage for Undergraduates* was finished, I found the Sage syntax to be a bit clumsy and awkward, and I thought that a few easy-to-implement features were missing.

Yet, that's the beauty of open-source software. When you see something that you think is sub-optimal, or simply missing, then you can reprogram it yourself! Therefore, I did exactly this, with the help of Harald Schilly of SageMath Inc. I wrote some new code, got some feedback, made changes, got more feedback, and Sage was improved by this process. That simply wouldn't be possible with a commercial product. How could I change one of the expensive closed-source computer-algebra packages?!

This code was written in the middle of August 2016. It takes time for the full peer-review to take place. Given the importance of mathematical software in general, and Sage in particular, I'm sure that you understand that the review process is rigorous and time-consuming.

The new code is available in several places.

- At `www.gregorybard.com` under "Sage Stuff."
- As Section H.13 on Page 1076.
- At the URL:
  `https://cloud.sagemath.com/projects/9107841b-69fe-4d5e-a16f-f3eb2468b07c/files/Newer%20Stuff/Newer%20Bits%20of%20the%20Graphical%20Appendix/new_plot3d_syntax.sage`
- Or at this URL: `https://cloud.sagemath.com/projects/9107841b-69fe-4d5e-a16f-f3eb2468b07c/files/Newer%20Stuff/Newer%20Bits%20of%20the%20Graphical%20Appendix/new_plot3d_syntax.sage`

H.1.2. Using the New 3D Plotting Command

As I noted above, my new plotting commands are not yet part of Sage, as of August of 2016. Therefore, you need to specifically import my code in order to use the command `new_plot3d`. In order to do this, it depends if you are working in SageMathCloud, or SageMathCell.

SageMathCell: You must copy the code for `new_plot3d` from one of the sources listed above, and paste it into your SageMathCell. Then, write your code below it. This is relatively inconvenient, but not too hard.

SageMathCloud: First, get the file `new_plot3d_syntax.sage` and place it in the same folder on SageMathCloud as your worksheet. Second, put the following commands toward the start of your SageMathCloud worksheet.

```
%auto
load("new_plot3d_syntax.sage")
```

These commands will execute automatically when the worksheet is opened, so they do not have to be in the first cell. Then you can use the new plotting syntax.

This situation is temporary. By early 2017 at the latest, the new commands should be a core part of Sage, and no special action will be required to use these commands.

Let's start with

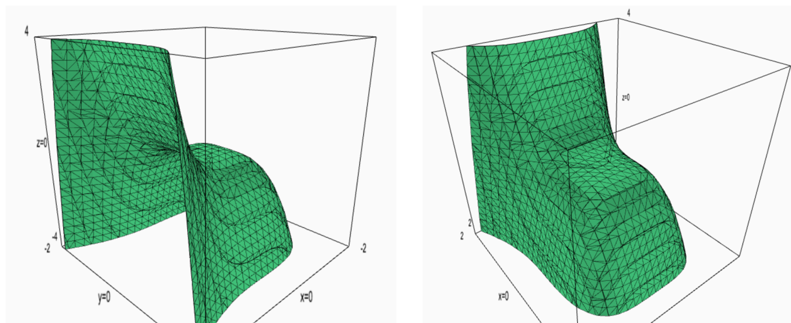
$$z = f(x, y) = x^3 - y^4$$

and plot this in a window given by $-2 < x < 2$ and $-2 < y < 2$. We might want to set $-4 < z < 4$, for example. Here is the required code:

```
var("x y")
f(x,y) = x^3 - y^4
new_plot3d( f, -2, 2, -2, 2, -4, 4 )
```

The image produced can be dragged around and rotated, allowing the user to view the object from many different angles. It is often very important to rotate the object, so that it can be seen from several viewpoints—only then can the more complicated objects be understood.

Here are two different views:



Warning:

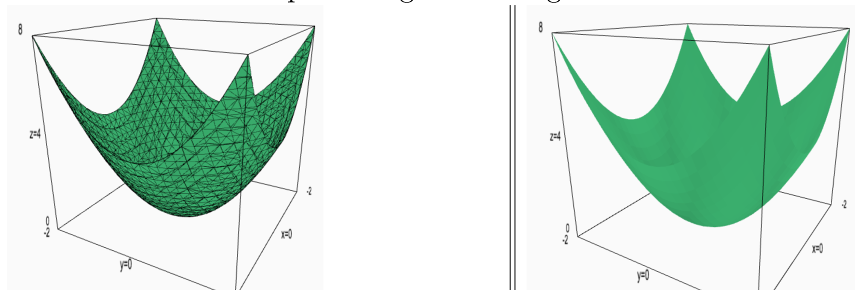
The 3D plots take up a large amount of memory and processor time. Therefore, it is best to limit yourself to two, three, or at most four of these plots on a single worksheet, when using SageMathCloud. Otherwise your project will consume too many resources, and your worksheet will begin to get very slow. When using SageMathCell, you should limit yourself to one 3D plot.

H.1.3. Options for the New 3D Plot Command

The dark mesh on the green object above helps the user visualize the object being displayed. However, sometimes it is "too much," interfering with the eye's ability to process the object. Here is a paraboloid, shown from a nice angle, both with and without the mesh.

$$z = f(x, y) = x^2 + y^2$$

as well as the code for producing those images.

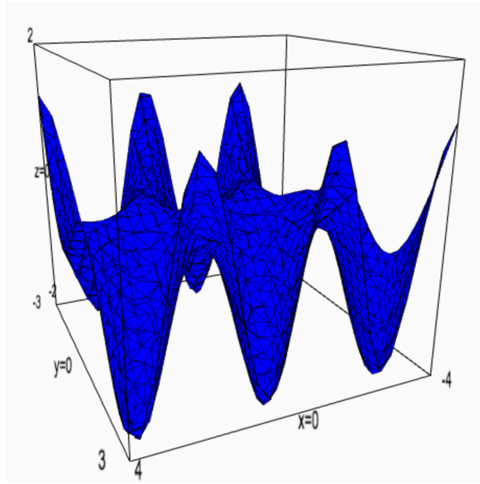



```
var("x y")
f(x,y) = x^2 + y^2
new_plot3d( f, -2, 2, -2, 2, 0, 8 )
```

```
var("x y")
f(x,y) = x^2 + y^2
new_plot3d( f, -2, 2, -2, 2,
          0, 8, mesh=false )
```

As you might have guessed, the mesh is turned on by default, but can easily be turned off by `mesh=false`.

We can also change the color of the plot, using the `color` optional parameter. Here's a nice wavy surface:



The code for that wavy surface is below.

```
var("x y")
new_plot3d( sin(x-y)*y*cos(x), -3, 3, -3, 3, -1, 1 )
```

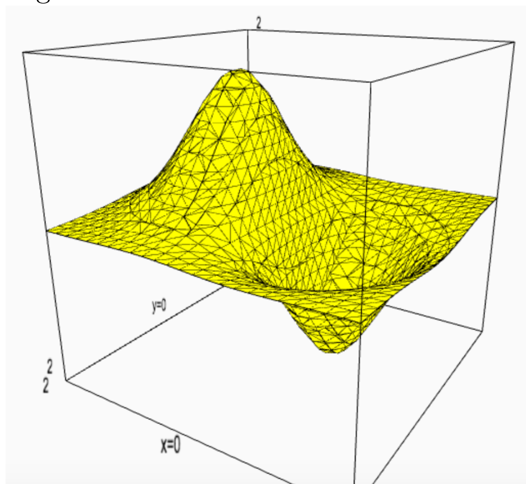
Next, we can plot

$$z = f(x, y) = 4xe^{-x^2-y^2}$$

using the following code:

```
var("x y")
new_plot3d( 4*x*exp(-x^2-y^2), -2,2, -2,2, -2,2, color="yellow" )
```

Here is the image:



By the way, in Section H.1.7, we'll see that function plotted again, but as a table-cloth plot.

Questions of Resolution Density

Let's suppose that we wanted to see a plot of the function

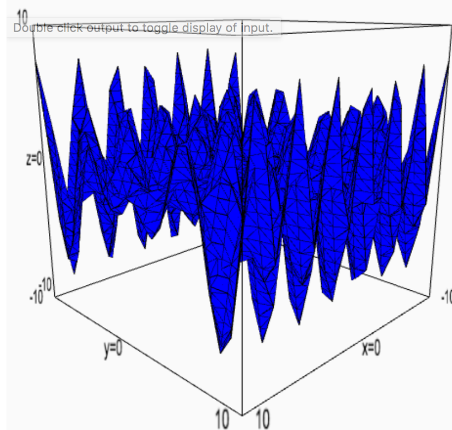
$$z = f(x, y) = (\sin(x - y))(y)(\cos x)$$

for the interval $-10 < x < 10$ and $-10 < y < 10$ as well as $-10 < z < 10$.

Based on what we know so far, we would type the following code:

```
var("x y")
new_plot3d( sin(x-y)*y*cos(x),  -10, 10, -10, 10, -10, 10,
      color='blue')
```

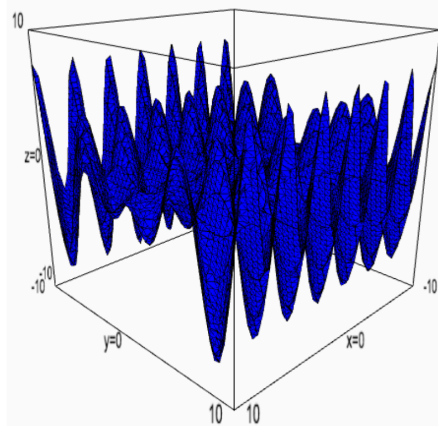
We obtain the follow image, which isn't completely satisfactory.



As you can see, the plot looks jagged, instead of smooth. It doesn't really convey the mathematical "truth" of the surface. By default, the new syntax divides each axis into 25 sub-intervals, creating $25^3 = 15,625$ small cubes. Most of the time, that's sufficient, but in this case it was not. We're going to add the optinal parameter `plot_points=50` now. The code becomes

```
var("x y")
new_plot3d( sin(x-y)*y*cos(x),  -10, 10, -10, 10, -10, 10,
      color='blue', plot_points=50 )
```

This now results in the following, much improved plot:



It should be noted that $50^3 = 125,000$ versus $25^3 = 15,625$ cubes are being used to generate the plot. That's a significant difference—$8\times$ as many cubes. That's why it takes so much longer.

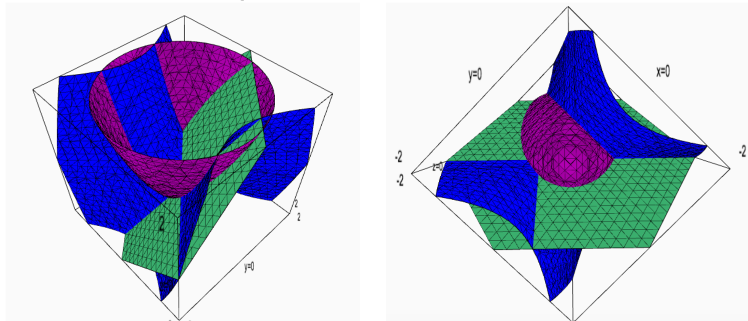
Going above 50 will either take so long that a "time out" occurs, or the file will be too large for SageMathCloud to transfer. For situations like this, you can use the old syntax. In fact, we'll re-examine this same function again on Page 1047, but with `plot_points=75`, in the old syntax.

Superimposition and Opacity

Just as the addition operator for 2D plots allowed for superimposing plots (see Section 1.4.2), the same is true for 3D plots. This can create some stunningly interesting images. For an example, try the following code:

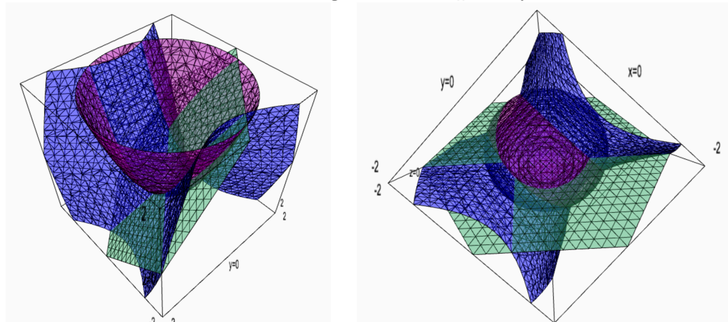
```
var("x y")
P1 = new_plot3d(x^2 - y^2,  -2,2,-2,2,-2,2, color='blue')
P2 = new_plot3d(x^2 + y^2 - 1,  -2,2,-2,2,-2,2, color='purple')
P3 = new_plot3d(x + y,  -2,2,-2,2,-2,2 )
P=P1+P2+P3
show(P)
```

Here is the image obtained, from two convenient viewpoints.



If we add the option `opacity=0.5` to each `new_plot3d` command, then we'll make the objects partially translucent (or "see through.") An opacity of 0.3 is rather translucent, but a 0.5 somewhat less so; an opacity of 0.8 is essentially solid, and an opacity of 1.0 is absolutely solid. Normally, when working with 3D plots, the opacity defaults to 1.0 unless otherwise changed.

Here are the same images with opacity set to 0.5.



The Size on the Screen:

Sometimes, when I'm making a presentation, I'd really like the 3D plot to be huge. By default (at least on my laptop), it covers about 1/3 the screen vertically, and about 1/9 horizontally. This is so that you can still see other parts of your SageMathCloud worksheet, and continue working.

If you want a large image, then you can do the following trick. Instead of typing this code:

```
plot3d( x^4 - 5*x^2 + 4 + y^2, (x,-2.5,2.5), (y,-2.5,2.5) )
```

you can type the following code instead:

```
show( plot3d( x^4 - 5*x^2 + 4 + y^2, (x,-2.5,2.5),
          (y,-2.5,2.5) ), width=1300 )
```

This gives you control over the width of the 3D-plot as it appears on your screen, in contrast with the length, width, and height of the actual mathematical object itself. By the way, the unit of measure in that case is "pixels." The new 3D object will be 1300 pixels wide.

Last but not least, it is worth mentioning that this trick will work with any plot in Sage, whether it be 2D or 3D, so far as I am aware.

Aspect Ratios and 3D Plotting

The `new_plot3d` command forces the 3D plot to appear in a cube. It does this by setting the aspect ratio according to the following formula:

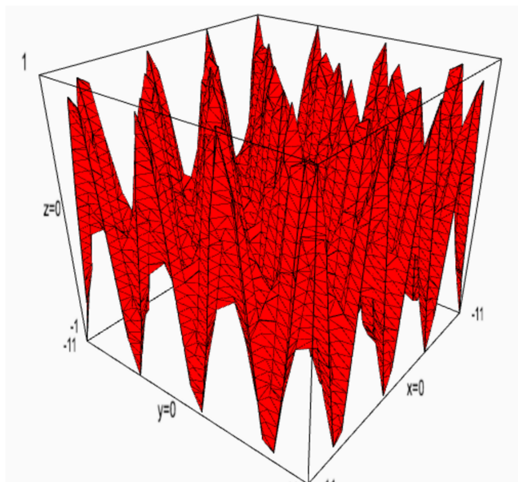
$$\left[\frac{1}{x_{\text{max}} - x_{\text{min}}},\ \frac{1}{y_{\text{max}} - y_{\text{min}}},\ \frac{1}{z_{\text{max}} - z_{\text{min}}}\right]$$

Usually, this is a good idea—but sometimes it creates a highly distorted image. Consider plotting $f(x, y) = (\sin x)(\sin y)$ for $-11 < x < 11$ and $-11 < y < 11$. As we all know, the output of sin varies from -1 to 1. We should not be surprised to see a highly distorted image, because the x-interval and y-interval plotted are 22 units long, whereas the z-interval plotted is only 2 units long.

If we type the code:

```
var("x y")
new_plot3d( sin(x)*sin(y), -11,11,-11,11,-1,1, color='red' )
```

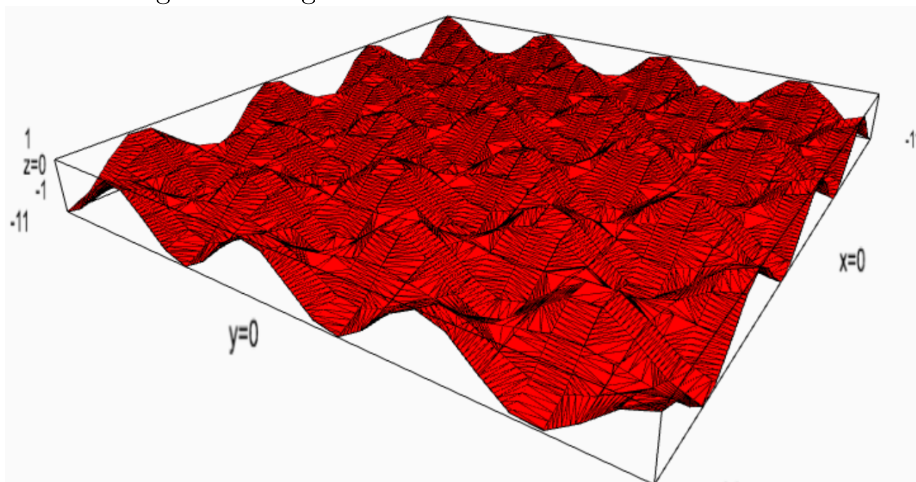
Then we get the image:



Instead, we can over-ride the default aspect ratio. For example, if we type the code:

```
var("x y")
new_plot3d( sin(x)*sin(y), -11,11,-11,11,-1,1, color='red',
                aspect_ratio = [1, 1, 1] )
```

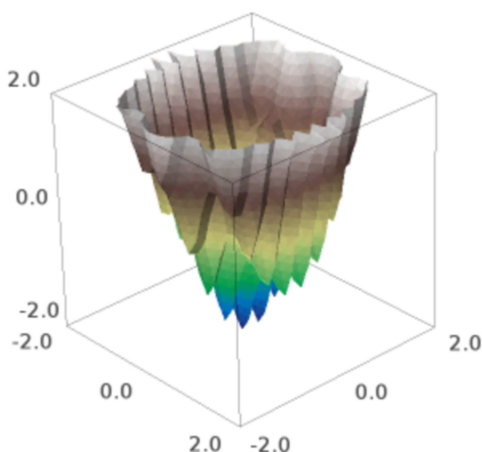
Then we get the image:



As you can see, modifying the aspect ratio produces a plot that is more faithful to the mathematical realities of the function.

H.1.4. Using Color Maps to make Terrain Plots

Like many ideas in this appendix, it is best to start with an example. Consider the following image, where the colors are chosen to represent the z-coordinate.



The key to this process is that one needs a function $T(x, y, z)$ that will map every point in the three-dimensional space being plotted into the interval $0 < T(x, y, z) < 1$. Next, a color map is chosen. These are the same color maps as listed in Section G.3.4. That color map will provide a color for each point (x, y, z), based on the number that $T(x, y, z)$ provides.

You can choose any color map that you want. However, I chose `terrain` because it looks like the type of terrain you might see in a video game. The highest points are white and then gray (representing mountains), and then brownish-yellows and greens (representing hills and plains), followed by various blues (representing seas). The resulting image can be very attractive. It is important to turn the mesh off with `mesh=False`, to avoid ruining the image.

Here is the code for making the above image.

```
var("x y z")

T(x,y,z) = (z + 2) / 4

f(x,y) = x^2 + y^2 - sin(10*y)^2 - cos(3*x)^2

new_plot3d( f, -2, 2, -2, 2, -2, 2,
      color = (T(x,y,z), colormaps.terrain),
      mesh=False ).show( viewer='tachyon' )
```

Since I expect that $-2 < z < 2$ in my plot, then $0 < z + 2 < 4$, so that

$$0 < \frac{z+2}{4} < 1$$

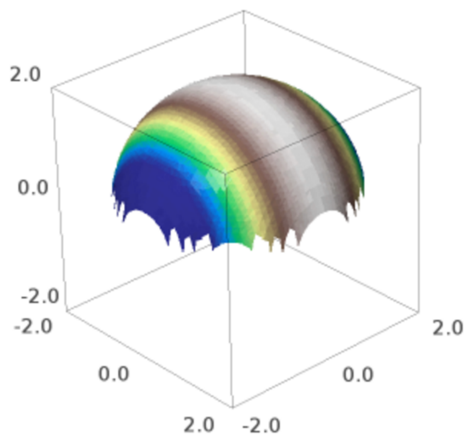
gives me the best choice for $T(x, y, z)$, if I want the color to represent z alone. However, there are many other ways to choose $T(x, y, z)$.

For another example, I chose $T(x, y, z) = \cos y$. Here is the code:

```
var("x y z")
```

```
T(x,y,z) = cos(y)

f(x,y) = sqrt(4 - x^2 - y^2)

new_plot3d( f, -2, 2, -2, 2, -2, 2, plot_points=50,
           color = (T(x,y,z), colormaps.terrain),
           mesh=false ).show( viewer='tachyon' )
```

The image produced is fairly attractive:



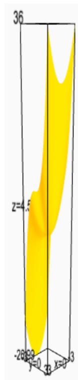
H.1.5. Comparing the Old and New Commands

Most readers will want to skip to Section H.1.7 on Page 1048 at this time. What follows is a comparison of the old `plot3d` command and the new command, `new_plot3d`. This is for you to have an idea of why the changes were made. Here are the reasons:

No Control over the z-window: While one could control the interval of x and the interval of y in the 3D plot with the old command, it was impossible to control the z-interval. You simply had to accept the fact that the graph plot would show all z-values achieved by the function over the plotted portion of the xy-plane.

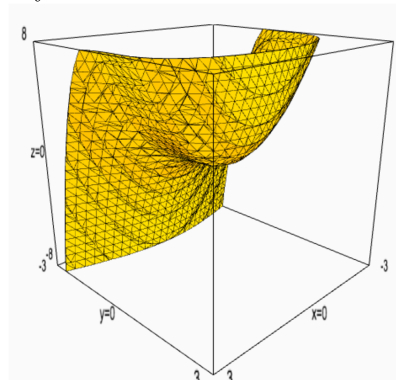
Default Aspect Ratio is 1,1,1: At first, it might not seem to be such a problem that the default aspect ratio was set to [1,1,1]. After all, one can override this with an optional parameter. However, that requires some knowledge of the function being plotted. The formula that I gave on Page 1041, to rapidly compute aspect ratios, requires $z_{\max}$ and $z_{\min}$. Often, students will want to see plots when first learning about functions of the form $z = f(x, y)$. They aren't ready, at that instant, to find all the global minima and maxima, and then find the true minimum and maximum on the plotted interval. In some cases, that's a lot of work.

Consider $z = f(x, y) = x^2 + y^3$. The following code plots it under the old syntax, and the new syntax.



The Old Syntax

```
var("x y")
plot3d( x^2 + y^3, (x,-3,3),
       (y,-3,3), color='orange')
```



The New Syntax

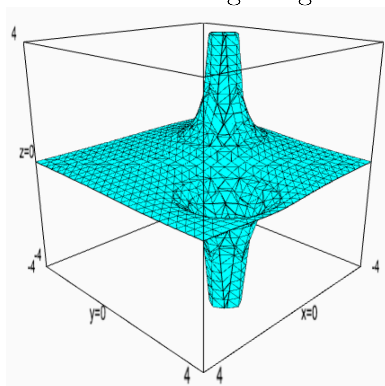
```
var("x y")
new_plot3d( x^2 + y^3, -3,3,
       -3,3,-8,8, color='orange')
```

Vertical Asymptotes are Impossible: One of the most frequently analyzed functions, either in complex analysis or in multivariate calculus, is a rational function. Any rational function with a vertical asymptote (in the portion of the xy-plane being plotted) would result in an unreadable plot. The z-values would take on an enormous interval, and then because the `plot3d` command uses an aspect ratio of [1,1,1], the x and y dimensions would be very tiny. This makes the plot look like a pen or pencil held vertically, and one can see nothing.

Rational functions work perfectly fine under the new syntax. For example, the following code:

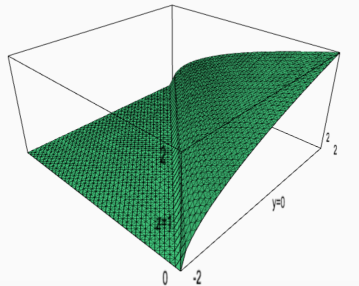
```
f(x,y) = 1/((x+2)^2 + (y+1)^2) - 1/((x-1)^2 + (y-2)^2)

new_plot3d( f(x,y), -4,4,-4,4,-4,4, color='cyan')
```

generates the following image:



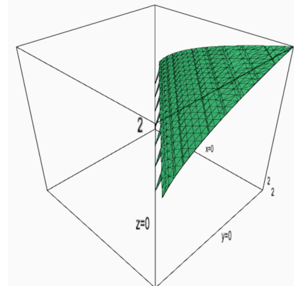
Undefined Values are Plotted as Zero: Sometimes there will be values that are undefined inside of the plotting range. The convention in mathematics is to leave such regions blank—completely unplotted. The old command does not do this, but the new command does. Instead, the old command plots $z = 0$ for those points.

For example, the graph of $f(x, y) = \sqrt{x + y}$ shows the value 0 if $x + y < 0$, which is not correct. The function takes on imaginary values when $x + y < 0$. Consider $f(-2, -2) = 2i \neq 0$. To see the effect visually, consider the following two 3D plots.



The Old Syntax

```
var("x y")
plot3d( sqrt(x + y), (x,-2,2), (y,-2,2),
        color='seagreen', mesh=True )
```



The New Syntax

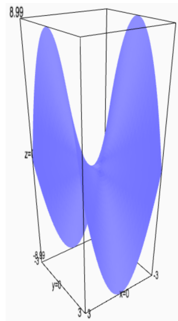
```
var("x y")
new_plot3d( sqrt(x + y), -2,2,-2,2,-2,2 )
```

Default Precision Too High: The default value for `plot_points` was set to 50, not 25. While this is useful in some cases, as we have already seen on Page 1039, it is usually unneeded. Moreover, this "wasted" precision can make the plot much slower to generate.

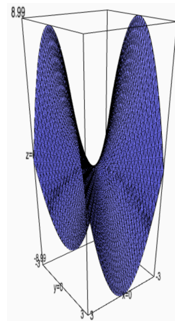
Mesh Off by Default: While this is far from a major issue, a 3D plot looks much better with the mesh than without, in the vast majority of cases. In the old syntax, the default was `mesh=False`. While some graphs look better that way, most do not. In the new syntax, the default is `mesh=True`. As an additional example, consider

$$z = f(x, y) = x^2 - y^2$$

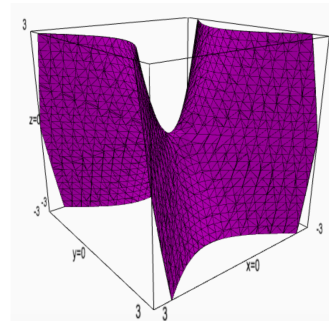
for $-3 < x < 3$ and $-3 < y < 3$.



Old Syntax
Mesh False by Default



Old Syntax
Mesh set to True



New Syntax
Mesh True by Default

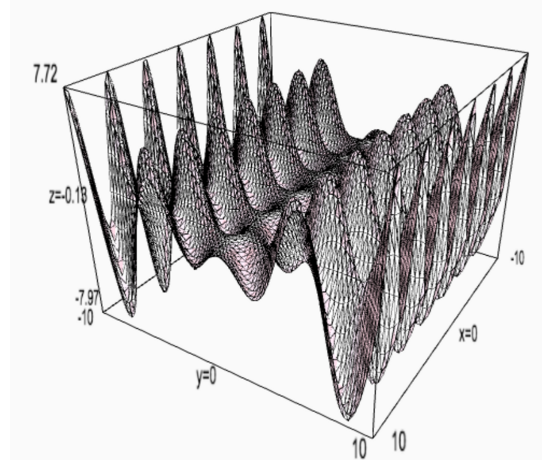
The old command had three advantages over the new command. First, it was much faster. This has to do with the internals, and it would be tedious to explain at this time. It can be summarized by saying that instead of dividing three-dimensional space into n^3 small cubes, it divides the xy-plane into n^2 small squares. Second, because it is so much faster, for very wobbly plots with a lot features, you can do `plot_points=75` or even `=100`. The third advantage is that the old command had the ability to do what I call "tablecloth" plots, which will be explained in Section H.1.7.

H.1.6. Using the old `plot3d` Command

The examples of the previous subsection give you an idea of how the syntax of the old command worked. The options `mesh`, `color`, `opacity`, and `aspect_ratio` work identically for both the old and new syntax. You can superimpose 3D plots, using the addition symbol, identically in both the old and new syntax.

Remember, you do not provide z-coordinates in the old syntax—Sage will compute those for you, regardless if they are convenient or inconvenient. If you need to compute an aspect ratio to make your plot fit in a cube, then the formula from 1041 will work. You can also multiply or divide one of the entries by two or three to get some nice rectangular boxes. (You can even use the golden ratio, if you are so inclined.)

Sometimes, when a graph varies a lot or is extremely wobbly, it is better to use the old syntax. For example, since the old syntax is so much faster, using `plot_points=75` or even `=100` should not be a problem. The following is the same plot we saw on Page 1039, except in pink instead of dark blue, using the old syntax, and with `plot_points=75`. As you can see, it looks really smooth now.



The code to produce that is below.

```
var("x y")
plot3d( sin(x-y)*y*cos(x), (x,-10,10), (y,-10,10), color='pink',
```

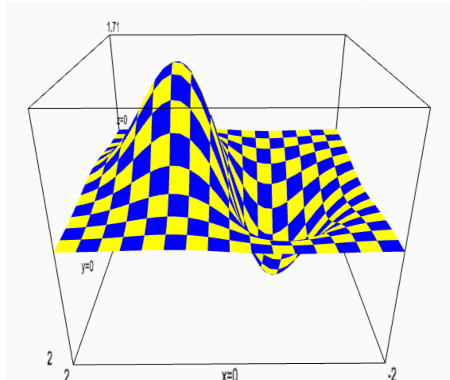
```
        mesh=True, plot_points=75 )
```

Note, I really do have to say `mesh=True`, because by default it is `False` in the old syntax. You might be wondering why the old syntax can handle `plot_points=75` when the new syntax has difficulty with `plot_points=50`. Instead of dividing three dimensional space into $75^3 = 421,875$ small cubes, the old syntax will divide the xy-plane into $75^2 = 5625$ small squares. That's why the old syntax is so much faster.

Last but not least, there is an option `adaptive=True`, which calls some extra code to make certain plots appear nicer. This also slows down the plotting. It rarely seems to be necessary, though it is necessary for the table-cloth plots which I am about to describe below.

H.1.7. Table-Cloth Plots

This concept is best explained by example. The following image:



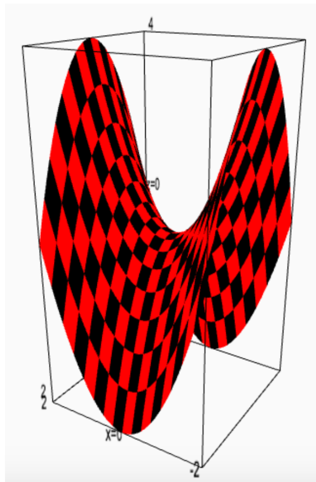
shows the function

$$f(x, y) = 4xe^{-x^2-y^2}$$

in what I call a table-cloth plot. (We saw this function before, on Page 1038.) The plot above was generated by the following code.

```
plot3d( 4*x*exp(-x^2-y^2), (x,-2,2), (y,-2,2),
        adaptive=True, color=['yellow', 'blue'] )
```

In general, you'll get a table-cloth plot when producing a 3D plot using `plot3d` (the old syntax), if you set `adaptive=True`, and provide a list of exactly two colors. Here is another view of $f(x, y) = x^2 - y^2$ from the previous subsection, but this time, it is a table-cloth plot.



H.2. Plotting Implicit 3D Surfaces

Sometimes, a multivariable function is defined implicitly, instead of explicitly. What I mean is that instead of $z = f(x, y)$ one could have some polynomial, for instance, in x, y, and z equal to zero. An example might be

$$4x^2(x^2 + y^2 + z^2 + z) + y^2(y^2 + z^2 - 1) = 0$$

This should not be too surprising. After all, while most functions in the univariate calculus are defined as $y = f(x)$, on the other hand, we write a circle like $x^2 + y^2 = 4$, or alternatively $x^2 + y^2 - 4 = 0$ as a polynomial in x and y, set equal to zero.

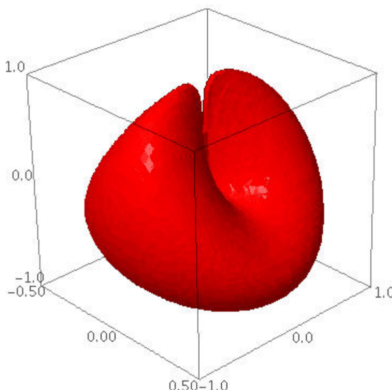
In any case, here's what you would do in Sage. First, we define the function that we wish to draw:

```
f(x,y,z) = 4*x^2 *(x^2+y^2+z^2+z) + y^2*(y^2+z^2-1)
```

Now the following command will plot the set of points where $f(x, y, z) = 0$, and the variables range over the values $-1/2 < x < 1/2$ and $-1 < y < 1$ as well as $-1 < z < 1$.

```
implicit_plot3d(f, (x,-0.5,0.5), (y,-1,1), (z,-1,1), color='red')
```

That produces this:



Before we wrap up our discussion of 3D plots, it is important to point out that if you were to use `plot_points=100` in the above command, then the x-interval would be divided into 100 parts, as well as the y and z intervals, for a total of

$$100 \times 100 \times 100 = 1,000,000$$

plotting points. This is extremely unwise, and can crash your web browser. Generally, I would not recommend going above 50. This point is discussed in more detail on Page 1040.

H.3. Plotting 3D Polyhedra

Before we move onto 3D plotting in general, it is nice to warm up a bit with a simple case, that of polyhedra. A polyhedron is like a polygon, except that it is in 3D instead of in 2D.

H.3.1. Built-In Polyhedra, the Platonic Solids

The following single commands, typed along either into a SageMathCell or into SageMathCloud will produce a nice pleasing figure. This can be useful for three reasons. First, it is a quick way to verify if 3D graphics are functioning on your particular setup and configuration. Second, it will help you get accommodated to the finger movements on your trackpad (or mouse) for rotations and scaling. Third, if you just want to demonstrate Sage's 3D graphics capabilities, then this is a nice way to do it. These are sorted in order from "most cool" to "least cool," in my humble opinion.

- `dodecahedron()`
- `icosahedron()`
- `octahedron()`
- `tetrahedron()`
- `cube()`
- `sphere()`

You can also make a multicolored cube, which looks very nice:

```
cube(color=["red", "black", "yellow", "green", "blue", "purple"])
```

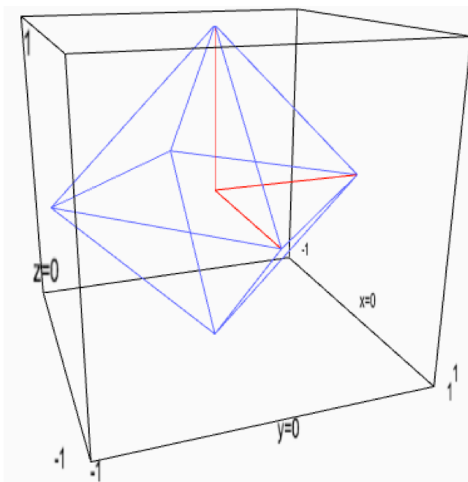
Historical Context (Ancient Greece)

The first five of those are the Platonic solids. While Plato of Athens was known more for his philosophical, political, and legal writings, it seems that he did have respect for geometry. He identified these platonic solids with the five elements of atomic theory, as it was known at that time. The icosahedron, or d20, was associated with water; the dodecahedron, or d12, was associated with aether; the octahedron, or d8, was associated with air; the cube (hexahedron) or d6 was associated with earth; the tetrahedron, or d4, was associated with fire.

You can find a discussion of this in Plato's dialog called $T\iota\mu\alpha\iota o\varsigma$ (Timaeus), but be sure to get a modern translation. Even slightly older translations are nearly impossible to read because the translators sought to beautify the writing by using archaic English. This is the same dialog where the legend of "the lost city of Atlantis" entered (so far as we are aware) into Western culture. These five polyhedra are the only polyhedra in three-dimensional geometry where each face is an identical, equiangular, and equilateral polygon.

H.3.2. An Example: A Skeleton of an Octahedron

The code in Figure 1 on Page 1053 will provide an extended example for us. In the classroom, this is an excellent way to introduce three-dimensional coordinates, and how they work. It will draw the skeleton (the edges) of an octahedron in blue, and then the positive x, y, and z axes in red. You get the following image:



On a printed page, or as a static image on the screen, the image is not perfectly comprehensible. However, if students are allowed to rotate the figure, and see it from many different angles, then it becomes entirely

comprehensible. This really shows how the ability to rotate and scale an image aids student comprehension.

The code shows you how we define a 3D coordinate (an ordered triple) in Sage. The `line` command will connect a list of points with line segments, constructing a polygon. (This might annoy some geometry professors, as line does not draw a line, nor a line segment, but a set of line segments.) If you'd like the enclosed area to be a closed polygon, you must list the start pointing both at the start of the list and at the end. Here, our polygons that represent the faces of the octahedron are triangles, so we have to list 4 points.

As in other types of Sage plotting, the addition operator `+` represents superimposition.

One trick that I like to do is replace

```
my_image = skeleton + axes
```

with instead

```
my_image=skeleton + axes + sphere(P_east, radius=0.1, color='green')
```

to break the symmetry, and identify the x-axis.

This circumstance is another opportunity to discuss Euler's equation for the relationship between the number of vertices, edges, and faces of a polyhedron. It is important to note that it works for all polyhedra without cavities, not just the five Platonic solids. With minor modification, you can even consider polyehdra with cavities.

A Challenge for You:

If you want to see if you have understood this material, then perhaps you should try to modify the above code to produce a cube.

H.3.3. The Polyhedron of a Linear Program

We saw earlier, on Page 1011 of Section G.2.2 of this appendix, that the feasible region of a particular system of inequalities in two variables was a convex polygon. We also stated that this will be true for all systems of linear inequalities in two variables, provided that the feasible region is bounded.

As you might guess, the analogous statement is true for systems of linear inequalities in three variables. The feasible region will be a convex polyhedron, provided that the feasible region is bounded. In this context, bounded means that there exists some sphere (regardless of where it is centered and of what radius) which entirely contains the whole feasible region.

```
P_up = (0, 0, 1)
P_down = (0, 0, -1)
P_north = (0, 1, 0)
P_south = (0, -1, 0)
P_east = (1, 0, 0)
P_west = (-1, 0, 0)
P_origin = (0, 0, 0)

face_1 = line( [P_up, P_north, P_east, P_up ] )
face_2 = line( [P_up, P_east, P_south, P_up ] )
face_3 = line( [P_up, P_south, P_west, P_up ] )
face_4 = line( [P_up, P_west, P_north, P_up ] )
face_5 = line( [P_down, P_north, P_east, P_down ] )
face_6 = line( [P_down, P_east, P_south, P_down ] )
face_7 = line( [P_down, P_south, P_west, P_down ] )
face_8 = line( [P_down, P_west, P_north, P_down ] )

pos_x_axis = line( [P_origin, P_east], color='red' )
pos_y_axis = line( [P_origin, P_north], color='red'  )
pos_z_axis = line( [P_origin, P_up], color='red'  )

skeleton=face_1+face_2+face_3+face_4+face_5+face_6+face_7+face_8
axes = pos_x_axis + pos_y_axis + pos_z_axis

my_image = skeleton + axes

my_image
```

Figure 1. An Extended Example, Drawing the Skeleton of an Octahedron. (For use with Section H.3.2 on Page 1051.)

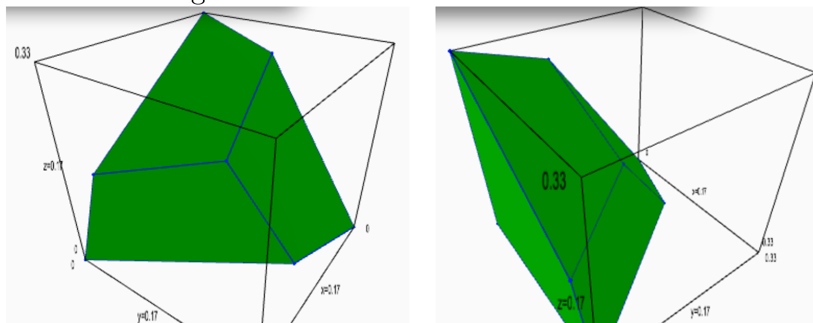
Sage can easily draw such a polyhedron, given a linear system of inequalities. Consider the following,

$$\begin{aligned} 2x_1 + x_2 + 3x_3 &\leq 1 \\ 3x_1 + 2x_2 + x_3 &\leq 1 \\ x_1 + 3x_2 + 2x_3 &\leq 1 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

Using the syntax that we learned in Section 4.21, for solving linear programs, we can write the following code. Note that we are talking about a region in 3-space that contains all the points which satisfy all the inequalities simultaneously. With that in mind, it is not surprising that we do not need to set an objective function, nor do we need to actually solve for the global feasible minimum or global feasible maximum.

```
p = MixedIntegerLinearProgram()

x = p.new_variable(real=True)

p.add_constraint(2*x[1] + x[2] + 3*x[3] <= 1)
p.add_constraint(3*x[1] + 2*x[2] + x[3] <= 1)
p.add_constraint(x[1] + 3*x[2] + 2*x[3] <= 1)

# these next two lines (and this comment) are not necessary.
# p.set_objective(x[1]+x[2]+x[3])
# p.solve()

my_polyhedron = p.polyhedron()
my_polyhedron.plot()
```

The above code produces the following polyhedron, displayed from two different camera angles.



H.4. The Best-Fit Plane

The Best-Fit Line shows up in high school science and mathematics classes, and therefore linear regression is familiar to college students when they see it. Then multivariate linear regression is presented, often with much trauma to the student. This can particularly be exacerbated by the fact that the students who are learning multivariate linear regression might be from social sciences like political science, nutrition, or management science—i.e. students who have no exposure to multivariate functions.

As a stepping stone between linear regression and multivariate linear regression, I suggest that the best-fit plane be presented. Because there are two explanatory variables and one output variable, the object can be graphed in 3-dimensional space. Consider the following comparison:

- For a best-fit line, we have dots (small circles) in a coordinate plane, and find a line that (1) goes through the average x and average y, (2) has the sum, over all data points, of the vertical distances to that line being zero, and (3) which minimizes the sum of the squares

of the vertical distances to that line. This gives a useful function

$$f(x) = \beta_0 + \beta_1 x$$

- For a best-fit plane, we have dots (small spheres) in three-space, and a plane that (1) goes through the average x, y, and z, (2) has the sum, over all data points, of the vertical distances to that plane being zero, and (3) which minimizes the sum of the squares of the vertical distances to that plane. This gives a useful function

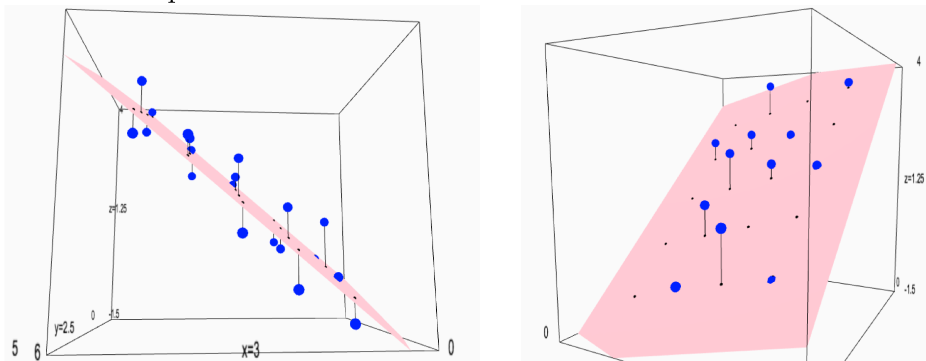
$$f(x, y) = \beta_0 + \beta_1 x + \beta_2 y$$

- For multivariate regression, we have

$$f(x_1, x_2, x_3, \ldots, x_n) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_n x_n$$

I think it is *extremely difficult* for a mathematically inexperienced student to pass directly from the first to the third. I know a solid professor, with a PhD in management science, who struggled with multivariate regression, presumably because of bad teaching—even though it is a favorite tool in graduate-level economics courses.

In the code given in Figure 2 on Page 1056, I show you my program and a sample (artificial) data set. The images below show the best-fit plane from different points of view.



Two Views of the Best-Fit Plane

You might be wondering how I found the best-fit plane in the first place. Using the techniques of Section 4.9 and Section 4.10, I typed the following:

```
var("beta_0 beta_1 beta_2")

our_data = [ (1, 1, 0), (2, 1, 2/5), (3, 1, 9/5), (4, 1, 2),
      (5, 1, 18/5), (1, 2, 4/5), (2, 2, 1/5), (3, 2, 9/5),
      (4, 2, 12/5), (5, 2, 14/5), (1, 3, -2/5), (2, 3, 1),
      (3, 3, 2), (4, 3, 12/5), (5, 3, 18/5), (1, 4, -6/5),
      (2, 4, -3/5), (3, 4, 2/5), (4, 4, 11/5), (5, 4, 11/5) ]

f(x,y) = beta_0 + beta_1*x + beta_2*y
find_fit( our_data, f )
```

```
var("x y z")

our_data = [ (1, 1, 0), (2, 1, 2/5), (3, 1, 9/5), (4, 1, 2),
      (5, 1, 18/5), (1, 2, 4/5), (2, 2, 1/5), (3, 2, 9/5),
      (4, 2, 12/5), (5, 2, 14/5), (1, 3, -2/5), (2, 3, 1),
      (3, 3, 2), (4, 3, 12/5), (5, 3, 18/5), (1, 4, -6/5),
      (2, 4, -3/5), (3, 4, 2/5), (4, 4, 11/5), (5, 4, 11/5) ]

dots = [ ]
for datum in our_data:
    dots.append( sphere(datum, color='blue', size=0.1) )

    z_hat = 0.85*datum[0] - 0.276*datum[1] - 0.5
    estimate = (datum[0], datum[1], z_hat)

    shadow = sphere( estimate, color='black', size=0.03 )

    tiny_line = line( [estimate, datum], color='black' )

    dots.append( shadow )
    dots.append( tiny_line )

display = sum( dots )

plane = implicit_plot3d( z == 0.85*x - 0.276*y - 0.5, (x,0,6),
     (y,0,5), (z,-3/2,4), color='pink' )

display+plane
```

FIGURE 2. My Program for Drawing a Best-Fit Plane

The model that Sage gave me was

$$z = -0.5 + 0.85x - 0.276y$$

so therefore $\beta_0 = -0.5$, $\beta_1 = 0.85$ and $\beta_2 = -0.276$.

H.5. Matrix Algebra and Intersecting Three Planes

When we have three linear equations in three unknowns, we can represent each equation as a plane floating in three-dimensional space. Much of the time, there is a unique solution. Students familiar with linear algebra (matrix algebra) will know that this will occur so long as the determinant is not zero.

However, other things can occur. There can be problems for which there is no solution. There can also be problems for which there are infinitely many

solutions. This further splits into two cases—one degree of freedom, and two degrees of freedom.

To help students visualize this, I have many interactive webpage that deals with this topic. Just go to

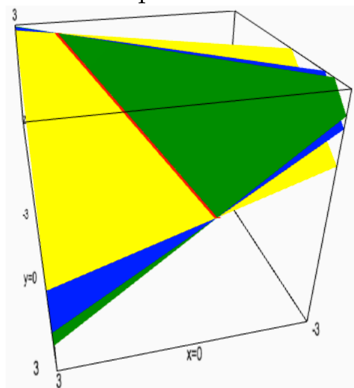
`www.gregorybard.com`

and click on "interactive webpages for teaching math." Then click on "Visualizing Infinitely Many Solutions in a Linear System of 3 Equations and 3 Unknowns." You can learn to make your own interactive webpages (or "apps" as they are often called) by reading Chapter 6 of *Sage for Undergraduates*.

The code for drawing a 3D-image of that type is given below. Here we are going to see a linear system with three equations, three variables, and one degree of freedom. The three planes will intersect not in a point, but in a line. Every point on that line is a solution to the linear system of equations. The line is drawn in red, whereas each of the planes is drawn in bright colors (yellow, green, and blue).

```
var("x y z")
# This is matrix C_2 in both Appendix D and Section 1.5

plane_1=implicit_plot3d(x + 2*y + 3*z==7, (x,-3,3), (y,-3,3),
          (z,-3,3), color='yellow')
plane_2=implicit_plot3d(4*x + 5*y + 6*z==16, (x,-3,3), (y,-3,3),
          (z,-3,3), color='blue')
plane_3=implicit_plot3d(7*x + 8*y + 9*z==25, (x,-3,3), (y,-3,3),
          (z,-3,3), color='green')

spine=line([ (-0.5,3,0.5), (2,-2,3) ], color='red', thickness=5)

show( plane_1 + plane_2 + plane_3 + spine )
```

That code produces the following image:

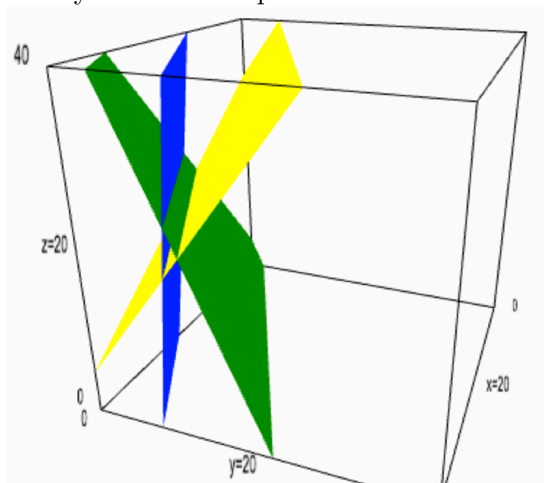


A Challenge for You:

The following linear system has no solutions. By modifying the above code, draw the image that represents the three planes and their lack of intersection.

$$\begin{aligned} 5x - 10y + z &= 115 \\ 2x - 4y + 3z &= 96 \\ 9x - 18y - 8z &= -38 \end{aligned}$$

The final image should vaguely look like this, but it depends upon what ranges of x, y, and z you choose to plot.



H.6. Plotting in Cylindrical Coordinates

Cylindrical coordinates are very important, particularly in Electrical Engineering. After all, wires (especially if you zoom in a lot) are cylinders. The workings of cylindrical coordinates are a minor extension of polar coordinates. For this reason, it might be useful to review Section 3.3, which dealt with polar plotting.

Basically, a cylindrical coordinate system can be thought of as someone who puts a piece of polar-coordinate graph paper, on a table. Points not on the table have a z-coordinate, that says how much above the table (if positive) or below the table (if negative) they are. Points on the table have $z = 0$. The point obtained by moving vertically until the table is contacted is on the graph paper, and has a polar coordinate given by (r, θ), just like any polar coordinate. In this way, any point in the universe can be described by three numbers in the format (r, θ, z). Just like in polar coordinates, θ should respect $0 \leq \theta \leq 2\pi$, but r can be any real number. Likewise, z can be any real number as well.

Just as curves in 2D are usually described as $f(x) = y$ and surfaces in 3D are often described as $f(x, y) = z$, surfaces in cylindrical coordinates are

defined by a radius function, $r(\theta, z)$. This function will output a radius for any given z-coordinate and θ-coordinate.

A particularly boring example might be $r(\theta, z) = 4$, which ignores both z and θ. This is going to generate a part of a cylinder, that looks to me like a soup can's label. That's not very exciting. The code for this is very simple, but note that z and θ have to be declared as variables with `var`, as do all variables except for x.

```
var("z theta")
# Cylindrical Example 1
cylindrical_plot3d( 4, (theta,0,2*pi), (z,-2,2),
          plot_points=[80,80])
```

You might be wondering what the `plot_points=[80, 80]` is about. The normal density of points (see Section H.1.3 on Page 1039) is insufficient for this figure. As before, as you increase the density, the rendering of the image takes longer, but the quality is higher.

Contrastingly, $r(\theta, z) = 2 + \sin(4z)$ will ignore θ, but not z. Such functions are symmetric about the z-axis and they happen to come up in calculus classes very often. This figure would have a radius varying sinusoidally with $1 \leq r \leq 5$, because

$$-1 \leq \sin(\text{anything whatsoever}) \leq 1$$

and inequality that we should all keep in mind. The code to produce this image is below.

```
var("z theta")
# Cylindrical Example 2
cylindrical_plot3d( 2 + sin(4*z), (theta,0,2*pi),
          (z,-2,2), plot_points=[80,80])
```

A much more interesting example is

$$r(\theta, z) = e^{-z^2}(\cos(4\theta) + 2) + 1$$

for $0 \leq \theta \leq 2\pi$ and $-2 \leq z \leq 2$. The code for that is below.

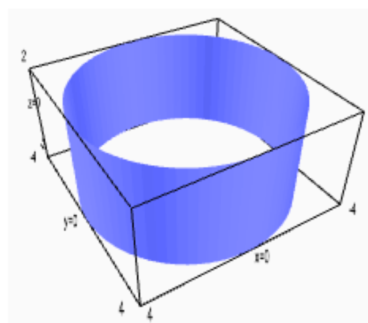
```
var("z theta")
# Cylindrical Example 3
cylindrical_plot3d( e^(-z^2)*(cos(4*theta)+2)+1, (theta,0,2*pi),
          (z,-2,2), plot_points=[80,80] )
```

A crazy example, but one that looks like a cool bit of exotic flora, can be given by

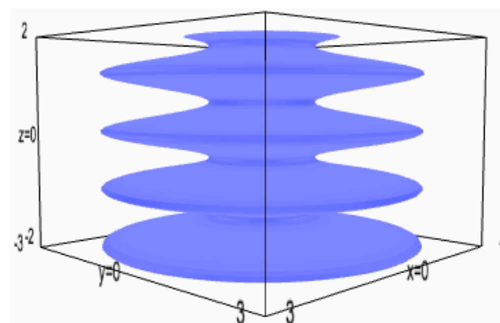
$$r(\theta, z) = (\sin 6z)(\cos 5\theta) + 1$$

again for $0 \leq \theta \leq 2\pi$ and $-2 \leq z \leq 2$. The code for this is below. You'll notice we went from 80^2 points to 200^2 points, because this figure is more complicated, and those extra points are needed to draw the figure properly.

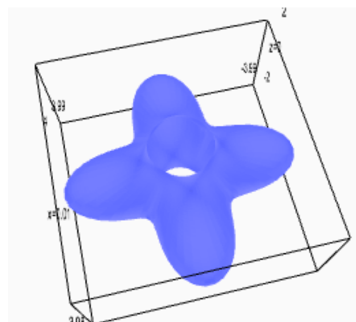
```
var("z theta")
# Cylindrical Example 4
cylindrical_plot3d( sin(6*z)*cos(5*theta)+1, (theta,0,2*pi),
          (z, -2, 2), plot_points=[200,200] )
```



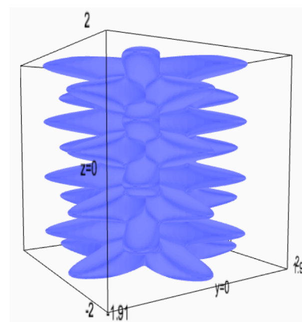
Example 1



Example 2



Example 3

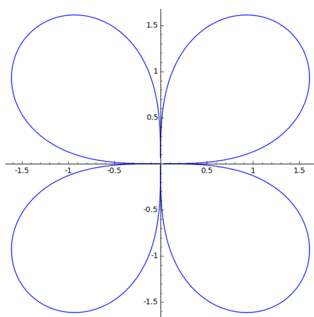


Example 4

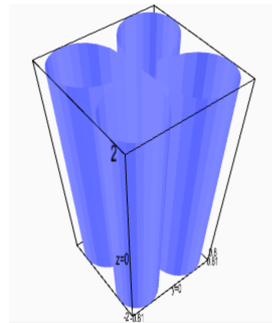
We stated earlier that when $r(\theta, z)$ ignores θ, then we have a figure which is symmetric around the z-axis, like a volume of revolution in *Calculus II*. You might be curious what happens if $r(\theta, z)$ ignores z. Then we're essentially back to polar coordinates, and we get what Plastics Engineering people call an extrusion. We can take an examples from Section 3.3, and see what happens if we extrude it.

For $r(\theta, z) = 2\sqrt{|\sin(2\theta)|}$ we would type

```
var("z theta")

cylindrical_plot3d( sqrt(abs(sin(2*theta))), (theta,0,2*pi),
          (z, -2, 2), plot_points=[80,80] )
```



Polar Version



Cylindrical Version

As you can see, it is as if we forced a bunch of liquid plastic through a form or filter that was shaped like the polar plot. This technique is frequently used in manufacturing.

Last but not least, I have been using sloppy language when I say "$r(\theta, z)$ ignores z" or "$r(\theta, z)$ ignores θ." What I really mean is that $\partial r/\partial z = 0$ or $\partial r/\partial \theta = 0$ as partial derivatives. However, what it really comes down to is that z or θ is simply missing from the simplified form of $r(\theta, z)$.

A Note about Aspect Ratio

There is another optional parameter, like `plot_points`, that can help your images. It is `aspect_ratio=[1,1,1]` and has to do with "drawing to scale." If the x-values vary from, perhaps, $1 < x < 4$ in your object, and the y-values vary from, perhaps, $1 < y < 1000$, then it is important that Sage *not draw* your graph to scale. Otherwise your graph would be almost invisible. However, when working with objects from geometry, it can be very important to "draw to scale," otherwise important characteristics, such as angles, will be distorted—sometimes distorted beyond recognition.

A Challenge for You:

Plot the function

$$r(\theta, z) = \left(0.1 + (\sin 5z)^2 (\sin 6\theta)^2\right) e^z$$

for $0 < \theta < 2\pi$, but $-7 < z < \pi/5$. This example is due to by Prof. Gilbert Labelle of the Université du Québec à Montréal.

H.7. Plotting Volumes of Revolution in *Calculus II*

Finding the volumes (and surface areas) of surfaces of revolution is a delightful topic in *Calculus II*, because we see the unity of geometry, ordinary algebra, and the integral calculus. It also helps us learn how to visualize in three dimensions—because sadly many of us only see two-dimensional geometry in high school. That's most unfortunate because we live in a three-dimensional world! Moreover, it is often hard to sketch in 3D, and it

can really help both faculty and students to see 3D graphics representing the objects whose volume or surface area they are trying to compute.

Cylindrical coordinates are the natural environment for producing these 3D images, because of how they are produced. The axis of symmetry will become the z-axis, and "the other axis" will become the r-axis. These figures are symmetric about the z-axis, and therefore θ will play no role in the radius function.

- If you're rotating around the x-axis, then x takes the role of z and y takes the role of r. This means that the equation to produce your radius function should start with $y =$.
- If you're rotating around the y-axis, then y takes the role of z and x takes the role of r. This means that the equation to produce your radius function should start with $x =$.
- Please, do not just memorize this. Honestly, try to understand why these two facts above (and the one below) is true.
- Another option is to do "off-axis" rotations. That means you are rotating around $y = 2$ or $x = 4$. For these, you follow the rules for which ever axis your line of rotation happens to be parallel to.

By the way, since we are talking about visualizing an actual geometric object, as compared to analyzing the relationships between variables, we really do need the aspect ratio to be controlled in these problems. Therefore, we will always use the optional parameter `aspect_ratio=[1,1,1]`.

Example One:

Let's consider, as our first example, rotating the curve $y = e^{x-1}$ about the x-axis for $0 \leq x \leq 3$. Because we are rotating around the x-axis, the role of r will be taken by y and the role of z will be taken by x. As mentioned before, θ does not play a role here because these sorts of figures are symmetric about the axis of rotation. There we should change the Cartesian-coordinates function

$$f(x) = y = e^{x-1}$$

into the cylindrical-coordinates function

$$r(\theta, z) = e^{z-1}$$

for $0 \leq z \leq 3$.

We should type

```
var("z theta")
cylindrical_plot3d( e^(z-1), (theta,0,2*pi), (z, 0, 3),
          aspect_ratio=[1,1,1] )
```

The plot is shown below. On the left, we have $0 \leq \theta \leq 2\pi$, which is the figure itself. On the right we have $0 \leq \theta \leq (3/2)\pi$, where only 75% of the figure is being drawn. This helps show how the figure is constructed.

Example 1
$0 \leq \theta \leq 2\pi$

Example 1
$0 \leq \theta \leq \frac{3}{2}\pi$

Example Two:

We're going to take the same curve as Example 1, namely $y = e^{x-1}$ for $0 \leq x \leq 3$, but now rotate it around the y-axis. Because we are rotating around the y-axis, the role of r will be taken by x and the role of z will be taken by y. As always, θ does not play a role here because these sorts of figures are symmetric about the axis of rotation. There we should change the Cartesian-coordinates function

$$\begin{aligned} y &= e^{x-1} \\ \log_e y &= x - 1 \\ 1 + \log_e y &= x \\ 1 + \log_e z &= r \end{aligned}$$

into the cylindrical-coordinates function

$$r(\theta, z) = 1 + \log_e z$$

for $0 \leq z \leq 3$. However, $\log_e 0$ is undefined, in the sense that

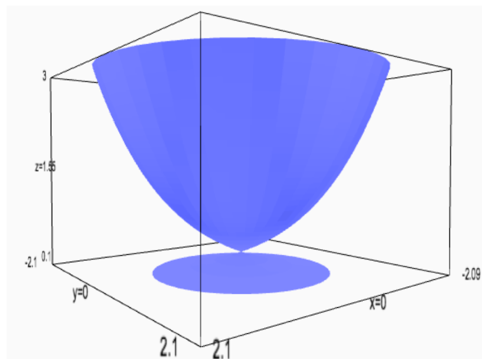
$$\lim_{x \to 0^+} \log_e x = -\infty$$

so we should use $1/10 \leq z \leq 3$. Alternatively, we could have chosen $1/100$ or $1/5$ for the lower-bound of z.

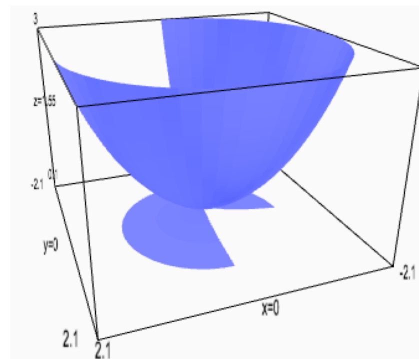
It is a good idea to note that $\log_e x$ is written $\ln x$ in most calculus textbooks, while it is `log(x)` in Sage, as discussed in Section 1.2, under the heading "Logarithms."

In any case, we should type

```
var("z theta")
cylindrical_plot3d( 1 + log(z), (theta,0,2*pi),
          (z, 1/10, 3), aspect_ratio=[1,1,1] )
```



Example 2
$0 \leq \theta \leq 2\pi$



Example 2
$0 \leq \theta \leq \frac{3}{2}\pi$

Example Three:

Now we're going to draw the area between $y = x^2$ and $y = x^3$, for positive x and positive y, rotated around the y-axis. This will require two cylindrical plots, one for the "inside surface" and one for the "outside surface." While we will not compute the volume here, we would be required to use the method that some books call "The Method of Washers" and which other books call "The Method of Annuli." This is in contrast with our previous two examples, which would require that some books call the "The Method of Pancakes" and which other books call "The Method of Discs."

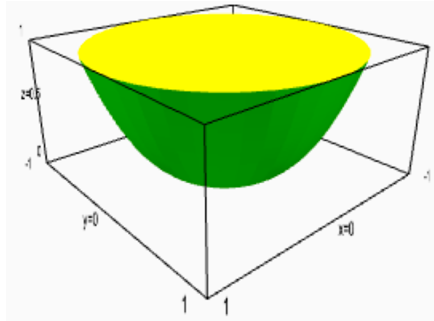
First, we have to find out where the two curves cross each other. Surely at the crossing point, the y-coordinates of the curves are equal.

$$\begin{aligned} y &= y \\ x^2 &= x^3 \\ 0 &= x^3 - x^2 \\ 0 &= x(x^2 - 1) \\ 0 &= x(x-1)(x+1) \end{aligned}$$

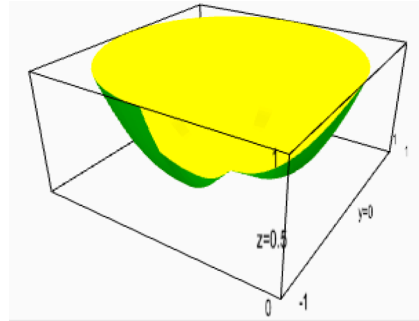
We get $x = 0$, $x = 1$, and $x = -1$. Since we have restricted to positive x, we know that we should use $0 < x < 1$. Coincidentally, the y-coordinates are also $0 < y < 1$. Now we're ready to plot. We're actually going to use 10^{-6} instead of 0, which might be a surprise. To see why, try the code below, and then try it again, but replacing the 10^{-6}s with 0s.

```
var("z theta")
inside = cylindrical_plot3d( z^(1/2), (theta,0,(3/2)*pi),
          (z, 10^-6, 1), color='yellow', aspect_ratio=[1,1,1] )
outside = cylindrical_plot3d( z^(1/3), (theta,0,(3/2)*pi),
          (z, 10^-6, 1), color='green', aspect_ratio=[1,1,1] )
inside + outside
```

Notice that we carried out the superimposing of the images, to see the inside surface and the outside surface simultaneously, through "adding" the two outputs from the `cylindrical_plot3d` commands.

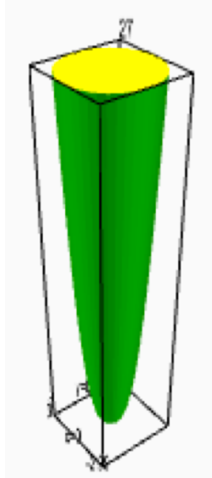


Example 3
$0 \leq \theta \leq 2\pi$

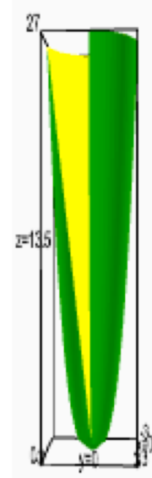


Example 3
$0 \leq \theta \leq \frac{3}{2}\pi$

A Challenge for You:



My challenge for you is to plot the volume of revolution formed by taking the area between $y = 9x$ and $y = x^3$, and rotating it around the y-axis. The image obtained is shown on either side of this paragraph.



H.8. Plotting in Spherical Coordinates

Many students find spherical coordinates confusing, but there is a trick to learning this useful tool. We'll start by exploring the coordinate system itself.

H.8.1. Introduction to Spherical Coordinates

For any point, the coordinates are (r, θ, φ), where r is the radius, and θ is the same as theta from polar coordinates. You can imagine a horizontal plane, representing $\varphi = 0$ as a straight-forward polar-coordinate plane. The trick is to fix some specific radius r, and then you get a sphere. That polar-coordinate plane will contain that sphere's equator.

You are probably familiar with longitude and latitude from geography. The θ is exactly the same as longitude. Just as on the earth, where longitude measures how far east or west you are compared to Greenwich in the United Kingdom, in this coordinate system, the θ represents moving in a horizontal[1] direction.

Since θ is longitude it is reasonable to presume that φ is latitude. Unfortunately, the φ is not latitude. Regrettably, many students think that it is latitude. Let's now explore what φ and latitude have in common and how they differ.

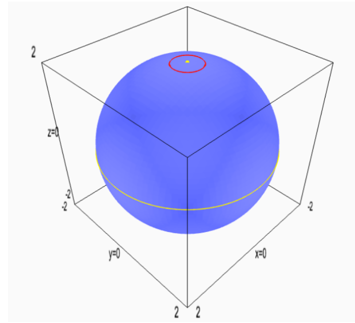
- In geography, the north pole is 90°N latitude. The equator is 0° latitude. The south pole is 90°S latitude. A short jog around the north pole is 89°N latitude, and a short jog around the south pole is 89°S latitude.
- The "north pole" in spherical coordinates is $\varphi = 0°$ or $\varphi = 0$ radians. The equator is $\varphi = 90°$ or $\varphi = \pi/2$ radians. The south pole is $\varphi = 180°$ or $\varphi = \pi$ radians. A short jog around the north pole is $\varphi = 1°$ or $\varphi = (1/180)\pi$ radians. Contrastingly, a short jog around the south pole is $\varphi = 179°$ or $\varphi = (179/180)\pi$ radians.

As you can see, ϕ and latitude are different, but they are highly related. With this in mind, it is not surprising that ϕ is called "the co-latitude." Interpreting north latitudes as positive, and south latitudes as negative, then it is always true that the latitude plus the co-latitude will equal 90° or $\pi/2$ radians. They are complementary angles, hence the use of the "co-" prefix.

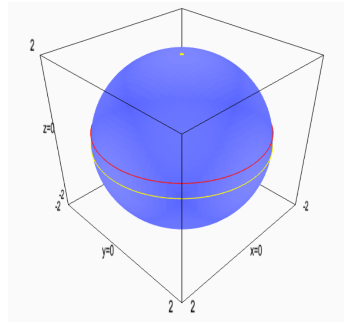
It should be noted that some textbook authors write (r, θ, φ) and some write (r, φ, θ). While the ordering is non-standard, and that surely is an annoying ambiguity, the symbols are consistent.

The following diagrams show a large blue planet, perhaps Neptune. The yellow dot at the top is the north pole. The yellow circle about the middle is the equator. On the left, the red circle is 80°N latitude but $\varphi = 10°$ or $\varphi = \frac{1}{18}\pi$ radians. On the right, the red circle is 10°N latitude but $\varphi = 80°$ or $\varphi = \frac{4}{9}\pi$ radians.

[1]To be precise, if you change only θ, then you are moving in such a way as to leave the distance to the north pole a constant, the distance to the south pole a constant, and the distance to the equator, a constant.



This is 80°N for latitude
This is $\varphi = 10^\circ$ or $\varphi = \frac{1}{18}\pi$



This is 10°N for latitude
This is $\varphi = 80^\circ$ or $\varphi = \frac{4}{9}\pi$

H.8.2. Examples of Plotting with Spherical Coordinates

Of course, if you'd like to see a portrait of a sphere, then you can choose

$$r(\theta, \phi) = 1$$

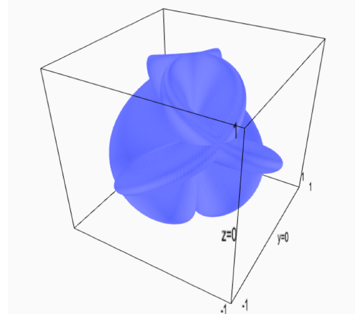
but usually we want something more complicated. Consider these

(1) $r(\theta, \phi) = \sin(2\theta + 3\phi)$
(2) $r(\theta, \phi) = (\sin 5\theta)(\cos 6\phi)$
(3) $r(\theta, \phi) = (\cos 4\theta)(\cos 3\phi)$
(4) $r(\theta, \phi) = 1 + \frac{1}{3}(\sin 6\theta)(\cos 5\phi)$

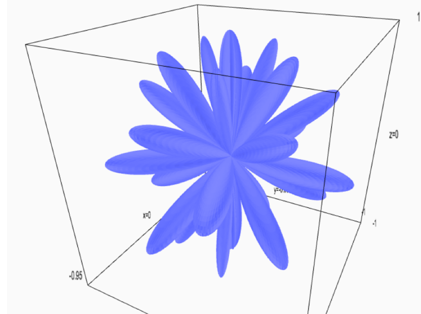
For example, to plot the first example, use the following code:

```
r(theta, phi) = sin( 2*theta + 3*phi )
spherical_plot3d(r, (theta,0,2*pi), (phi,0,pi), plot_points=[200,200])
```

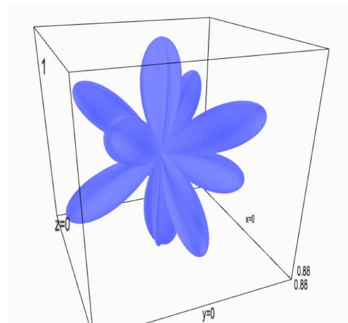
The four examples produce the following images.



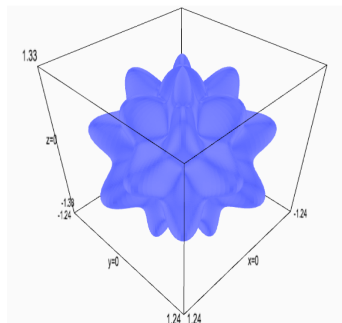
Example 1



Example 2



Example 3



Example 4

H.9. 3D-Parametric Space Curves and Derivatives

Both curves and surfaces in 3D space can be represented by writing three functions—one for the x-coordinate, one for the y-coordinate, and one for the z-coordinate. When we wish to model a curve, there is only one input to each function, usually denoted t. Therefore, the functions $x(t)$, $y(t)$, and $z(t)$, jointly define the curve. Historically, t was used because space curves were often used to represent the motion of a particle in three dimensional space. Contrastingly, 3D surfaces are represented parametricly by three functions of two variables: $x(u, v)$, $y(u, v)$, and $z(u, v)$.

Let's suppose, that at all times t, the position of a particle in 3-space is given by

$$x(t) = t^3 \qquad y(t) = t^2 \qquad z(t) = t$$

or alternatively

$$\vec{x}(t) = < t^3, t^2, t >$$

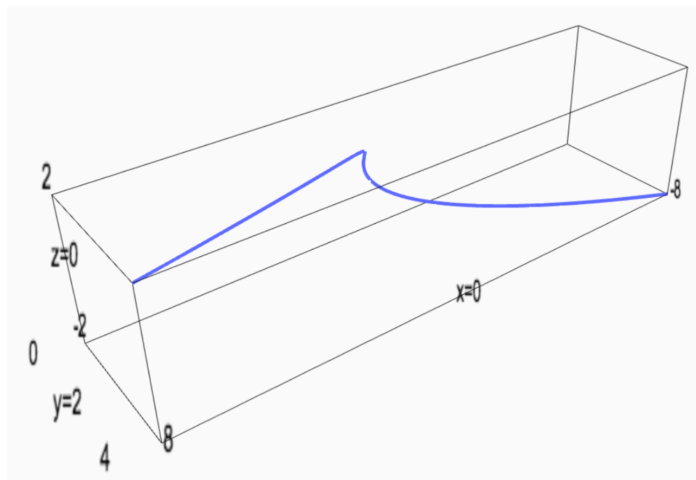
This can also be written a different way, commonly used in physics classes:

$$\vec{x}(t) = t^3\vec{i} + t^2\vec{j} + t\vec{k}$$

This is a classic space-curve, and is called "the twisted cubic." Here is the way that we would plot it in Sage, for the values $-2 \leq t \leq 2$.

```
x(t) = t^3
y(t) = t^2
z(t) = t
parametric_plot3d( [x(t), y(t), z(t)], (t,-2,2) )
```

Here is what the curve, the path of the particle, looks like. As you can see, it does "something interesting" at the origin.



Next, we might want to know the particles's velocity and acceleration. We can take the derivatives individually, and we would obtain the following:

$$x'(t) = 3t^2 \qquad y'(t) = 2t \qquad z'(t) = 1$$

as well as

$$x''(t) = 6t \qquad y''(t) = 2 \qquad z''(t) = 0$$

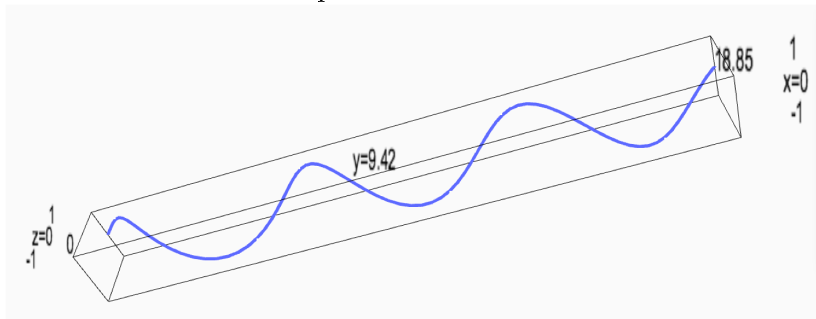
Velocity (being a vector) describes the direction of motion, and how fast the particle is moving in each coordinate axis. However, speed is often what we want—an indication of how fast the particle is moving in general, like the speedometer on a car. For speed, we would take magnitude of the vector, much like using the distance formula. We would the square root the sums of the squares of the entries of the vector. In our case, that comes to

$$||\vec{v}(t)|| = \sqrt{9t^4 + 4t^2 + 1}$$

and you can see that the speed is normally very high, but gets slow at the origin. The magnitude of the acceleration is also important at times, such as to measure the "g-forces" on a pilot or on the passengers of a roller-coaster. We will take the magnitude of the acceleration vector.

$$||\vec{a}(t)|| = \sqrt{36t^2 + 4}$$

Another classic example is "the helix." This is what it looks like



The functions are written below.

$$x(t) = \sin t \qquad y(t) = t \qquad z(t) = \cos t$$

That suggests the following code, for the values $0 \le t \le 6\pi$.

```
x(t) = sin(t)
y(t) = t
z(t) = cos(t)
parametric_plot3d( [x(t), y(t), z(t)], (t,0,6*pi) )
```

Again, we can compute the velocity and acceleration. We must take the derivatives individually, and we would obtain the following:

$$x'(t) = \cos t \qquad y'(t) = 1 \qquad z'(t) = -\sin t$$

as well as

$$x''(t) = -\sin t \qquad y''(t) = 0 \qquad z''(t) = -\cos t$$

Then we can find the speed

$$||\vec{v}(t)|| = \sqrt{(\cos t)^2 + 1^2 + (-\sin t)^2} = \sqrt{1+1} = \sqrt{2}$$

which is pretty neat. The acceleration function is also interesting

$$||\vec{a}(t)|| = \sqrt{(-\sin t)^2 + 0^2 + (-\cos t)^2} = \sqrt{1+0} = 1$$

This exposes an important misconception. While the derivative of the velocity function is the acceleration function, it is not true that the derivative *of the magnitude of the velocity* function is *the magnitude of the acceleration* function. Clearly,

$$\frac{d}{dt}\sqrt{2} = 0 \neq 1$$

and less obviously,

$$\frac{d}{dt}\sqrt{9t^4 + 4t^2 + 1} = \frac{36t^3 + 8t}{2\sqrt{9t^4 + 4t^2 + 1}} \neq \sqrt{36t^2 + 4}$$

I'm pretty sure that you will agree with me when I say that it is easier to take the derivative with your pencil, for these sorts of problems, than it is to even type these functions into Sage. However, for the sake of completeness, I should give the commands for computing the above work.

```
var("t", domain=RR)

position = vector( [ sin(t), t, cos(t) ]  )
velocity = diff( position, t )
acceleration = diff( velocity, t )
distance_to_origin = norm( position )
speed = norm( velocity )
norm_of_accel = norm( acceleration )

print "Position:", position
print "Distance to Origin:", distance_to_origin
```

```
print "Distance to Origin:", distance_to_origin.simplify_trig()
print
print "Velocity:", velocity
print "Speed:", speed
print "Speed:", speed.simplify_trig()
print
print "Acceleration:", acceleration
print "Magnitude of Acceleration:", norm_of_accel
print "Magnitude of Acceleration:", norm_of_accel.simplify_trig()

show( parametric_plot3d( position, (t,0,6*pi) ), width=1400 )
```

The code produces the following output:

```
t
Position: (sin(t), t, cos(t))
Distance to Origin: sqrt(t^2 + cos(t)^2 + sin(t)^2)
Distance to Origin: sqrt(t^2 + 1)

Velocity: (cos(t), 1, -sin(t))
Speed: sqrt(cos(t)^2 + sin(t)^2 + 1)
Speed: sqrt(2)

Acceleration: (-sin(t), 0, -cos(t))
Magnitude of Acceleration: sqrt(cos(t)^2 + sin(t)^2)
Magnitude of Acceleration: 1
```

There is also a plot that is shown after this output, but I've already included that plot above.

Challenge:
Modify the above code to draw this curve.

$$x(t) = \sin e^t \qquad y(t) = t \qquad z(t) = \cos e^t$$

for the values $-3 < t < 4$.

Here's a hint: you'll want to use a very high value for `plot_points`. For myself, I used 1000.

H.10. Plotting 3D Parametric Surfaces in Space

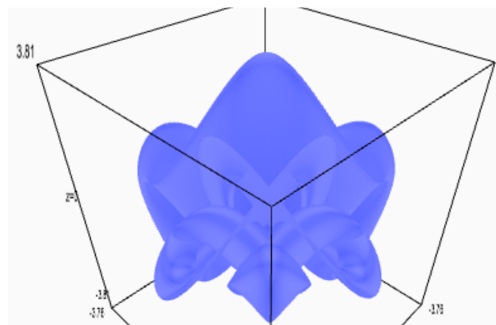
In a space curve, you have three functions. Each one is given a t, and responds back with either the x, y, or z coordinate. With a parametric surface, you again have three functions. This time, each one is given a u and a v, and as before, responds back with either the x, y, or z coordinate.

This can be made clear with an example.

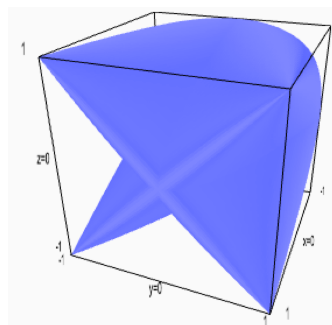
```
# The Devil's Mask
devil_x(u, v) = cos(2*u) + cos(v) + sin(u) + sin(2*v)
devil_y(u, v) = sin(2*u) + sin(v) + cos(u) + cos(2*v)
devil_z(u, v) = sin(2*u) + cos(3*v) + sin(2*v) + cos(3*u)
parametric_plot3d( [devil_x, devil_y, devil_z], (u,0,2*pi),
          (v,0,2*pi), plot_points=[200,200] )
```

Here's another example. The plots for both will be given further below.

```
# The Tic-Tac-Toe Stamp
my_x(u,v) = cos(2*u)
my_y(u,v) = cos(v)
my_z(u,v) = cos(u+v)
parametric_plot3d( [my_x, my_y, my_z], (u,0,2*pi),
          (v,0,2*pi), plot_points=[200,200] )
```



The Devil's Mask



The Tick Tac Toe Stamp

These particular cases were recommended to me by Prof. Josef Böhm, of The Austrian Center for Didactics of Computer Algebra, at the conference "Applications of Computer Algebra," held in Kalamata, Greece, in July of 2015. They come from an online collection of graphics by Prof. Gilbert Labelle of the Université du Québec à Montréal, which you can find at the following URL. Many other graphics in this appendix are from that source as well.

http://www.lacim.uqam.ca/∼gilbert

A Challenge for You:

Use the above code to plot the parametric surface

$$x(u,v) = \sin 2u \qquad y(u,v) = \sin 2v \qquad z(u,v) = (\sin u)(\sin v)$$

for the values $-\pi < u < \pi$ and $-\pi < v < \pi$.

H.11. 3D Vector Field Plots

A vector field plot is meant to represent a vector-valued function. The two-dimensional version will take two inputs, x and y representing a point on the Cartesian (ordinary) x, y-plane. For each such point, a two-dimensional vector is returned. Visually, this is represented by drawing an arrow at that point. The length of the arrow and size of the arrow head represents the magnitude of the vector, while the direction of the vector is usually more important, and is indicated by which way the arrow head is pointing. The two-dimensional version of this was covered in Section 3.7.

In three dimensions, for each point in space (x, y, z), we wish to draw a vector in three-dimensional space. Just like in the 2D case, the usual applications of the 3D case are to explore gravitational and electromagnetic fields. However, you can use these for any force field. Since our 2D example was electromagnetic, I thought it might be interesting for our 3D example to be about gravitation.

H.11.1. Example One: A Single Planet

Of course, we've all seen Newton's Law of Gravity in high school, or even middle school:

$$F = \frac{GM_{\oplus}m}{r^2}$$

where $M_{\oplus}$ is the mass of the earth, or some other planet; m is the mass of the satellite or other object that you are analyzing; r is the distance between the centers of the two objects; F is the (magnitude of the) force of gravity, and

$$G = 6.77 \times 10^{-11}\text{m}^3/(\text{kgsec}^2)$$

is the universal gravitational constant. Actually, we're using slightly sloppy language here. Usually we want to find how the gravity of a planet affects a satellite. However, you can use the same formula to discuss how two planets affect each other, or how a star affects a planet, or how to stars affect each other, and so forth.

That formula above gives us the magnitude of the force. The direction of the force is going to be the vector that starts at the satellite (has a tail at the satellite) and ends at the planet (has its head at the planet). However, because it is a direction-vector, we convert that vector into a unit vector—a vector of length one. The way that is done is that you divide the vector by its magnitude.

Therefore, we have a four step process.

- First, we find the vector which connects the object to the planet. Now we have to very careful about sign convention and direction here. Gravity always attracts and never repels. Thus we want the vector to point away from the satellite and toward the planet.
- Once that's done, we divide this "displacement" vector by its norm, and we get the direction, from the satellite's point of view, toward the planet.
- The magnitude is given by Newton's formula, but we've been sloppy here. We've decided that $GM_{\oplus}m$ is just 4. There exists a fictional set of units for which this is true. The magnitude of the displacement vector, normally called the distance, is r, so we divide by that magnitude squared to get the $1/r^2$ part of the formula.
- Finally, the final vector is the magnitude (a number or "scalar") times the direction vector. It is a scalar product (not a dot product, a cross product, nor what some textbooks call a box product and some textbooks call a triple product).

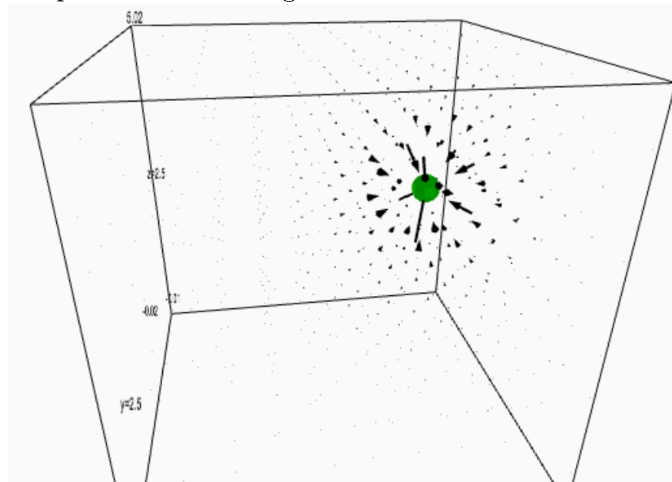
That force that we get from that four step process is the force to construct our force fields.

We use the `plot_vector_field3d` to make the vector field plot. We superimpose a sphere on the plot to show the location of the planet. That's done by adding the output of the `sphere` command to the output of the `plot_vector_field3d` command.

The code is below. The position of the planet is $< px, py, pz >=< 1, 2, 3 >$, and that's isolated at the top of the program to allow it to be changed.

```
px = 1
py = 2
pz = 3
k = 4

displacement = vector( [px-x, py-y, pz-z] )
direction = displacement / norm( displacement )
magnitude = k / norm(displacement)^2
force = direction*magnitude

planet = sphere( (px, py, pz), color='green', size=0.2 )

plot_vector_field3d( force, (x,0,5), (y,0,5), (z,0,5),
         center_arrows=True, plot_points=[10,10,10],
         colors='black', radius=0.02 ) + planet
```

That code produces this image.



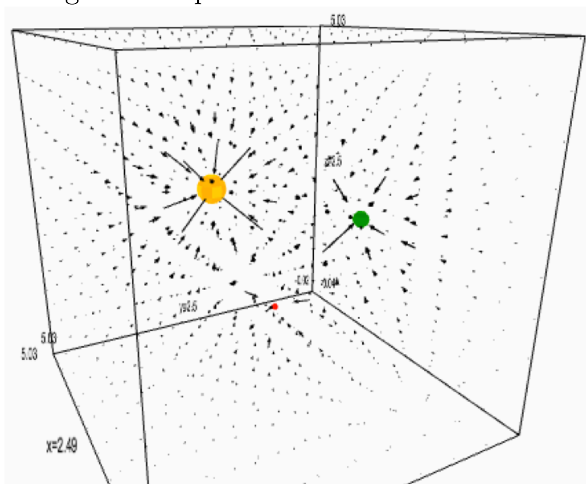
Here's a funny story. As it turns out, `len(displacement)` doesn't do what you think it does. While the magnitude of the displacement vector is the length of that vector, i.e. the distance between the two objects, that's not what the `len` command does. The `len` command comes from the computer language `Python`, upon which Sage is based. In `Python`, and therefore in Sage, the `len` command will return the number of entries in the list/vector. Here, that's just 3, because the vector consists of three quantities. Mathematicians call this "the dimension of the vector." The `norm` command gives us what we want, namely the magnitude of the vector.

H.11.2. Example Two: Three Planets

Now we've got three planets. An amazing fact from mathematical physics, of taken so much for granted, comes to our aid. If you have a bunch of forces acting on an object, and you want to find the net force, all you have to do is add them up. That's it. So this program will compute the force for three planets separately, each identical to our previous example—and then it adds them up.

Of course, we have to draw three spheres, and not just one. It helps to make them different colors and radii. Other than that, it an analogous program. The code is given in Figure 3 on Page 1077.

Here is the image that it produces.



H.12. Functions of a Complex Variable

Coming soon!

H.13. The New Plotting Code

```
# Wrappers for 3d Plotting, especially for complex valued functions
# License: GPLv3
# Copyright: 2016
# Authors:
# * Gregory Bard <gregory.bard@sagemath.com>
# * Harald Schilly <hsy@sagemath.com>
# accurate as of December 26, 2016.

def new_plot3d( f, xmin, xmax, ymin, ymax, zmin, zmax, **kwargs ):
    r"""
    This function provides a 3D plot of a function z=f(x,y) but in
    a restricted domain given by xmin < x < xmax, ymin < y < ymax, and
    zmin < z < zmax. It also computes the correct aspect ratio to make
```

```
p1x = 1
p1y = 2
p1z = 3
k1 = 4

p2x = 3
p2y = 2
p2z = 1
k2 = 3

p3x = 3
p3y = 3
p3z = 3
k3 = 10

displacement1 = vector( [p1x-x, p1y-y, p1z-z] )
displacement2 = vector( [p2x-x, p2y-y, p2z-z] )
displacement3 = vector( [p3x-x, p3y-y, p3z-z] )

direction1 = displacement1 / norm( displacement1 )
direction2 = displacement2 / norm( displacement2 )
direction3 = displacement3 / norm( displacement3 )

magnitude1 = k1 / norm(displacement1)^2
magnitude2 = k2 / norm(displacement2)^2
magnitude3 = k3 / norm(displacement3)^2

force1 = direction1*magnitude1
force2 = direction2*magnitude2
force3 = direction3*magnitude3

net_force = force1 + force2 + force3

planet1 = sphere( (p1x, p1y, p1z), color='green', size=0.1 )
planet2 = sphere( (p2x, p2y, p2z), color='red', size=0.05 )
planet3 = sphere( (p3x, p3y, p3z), color='orange', size=0.2 )

planets = planet1 + planet2 + planet3

plot_vector_field3d( net_force, (x,0,5), (y,0,5), (z,0,5),
          center_arrows=True, plot_points=[10,10,10],
          colors='black', radius=0.01 ) + planets
```

FIGURE 3. The Code that Solves the Example in Section H.11.2.

```
    the 3D plot appear in a cube. Note, if the widths of the intervals
    on x, y and z, are not the same length, then this will distort
    angles and some distances. Yet, this usually produces a very visually
    appealling 3D graph.

    Example:

    var("x y z")
    new_plot3d( x^3-y^4, -2,2,-2,2,-4,4 )

    The options color, plot_points, mesh, opacity, and aspect_ratio
    are available from implicit_plot3d(). Moreover, the options viewer,
    width, and possibly others are available from show().
    """

    assert xmin < xmax
    assert ymin < ymax
    assert zmin < zmax

    xwide = xmax - xmin
    ywide = ymax - ymin
    zwide = zmax - zmin

    # This line of code will see if mesh has been specified by the calling
    # code. If it has *not* been specified, it will be set to True.
    kwargs['mesh'] = kwargs.get('mesh', True )

    # This line of code will see if plot_points has been specified by the
    # calling code. If it has *not* been specified, it will be set to 25.
    kwargs['plot_points'] = kwargs.get('plot_points', 25)

    # This line of code will see if color has been specified by the
    # calling code. If it has *not* been specified, it will be set to seagreen.
    kwargs['color'] = kwargs.get('color', 'seagreen')

    # This line of code will see if the aspect_ratio has been specified by
    # the calling code. If it has *not* been specified, it will be set
    # to [1/xwide, 1/ywide, 1/zwide ]
    kwargs['aspect_ratio'] = kwargs.get('aspect_ratio', [1/xwide, 1/ywide, 1/zwide] )

    var("x y z")

    P = implicit_plot3d( z == f(x,y), (x,xmin,xmax), (y,ymin,ymax), (z,zmin,zmax), **kwargs )

    return P

def new_complex_plot3d( f, xmin, xmax, ymin, ymax, zmin, zmax, style, **kwargs ):
    """This plots a function whose sole input is a complex number, and whose sole
```

```
output is a complex number. There are many ways to plot such a function, and
they are chosen with the parameter style.

The options for style are 'magnitude', 'real', 'imaginary', 'argument', or 'mixed'.
Also 'phase' is a synonym for 'argument'.

Consider the function to be plotted as f(s) = f(x+iy). To be clear, the input to
the complex-function f is s=x+iy. The 3D plot has an x-axis, a y-axis, and a z-axis.

The x-axis is always the real part of the input, and the y-axis is always the
imaginary part of the input. The z-axis is determined by the choice of style.

For 'magnitude' it is the magnitude or norm of the output of f(s) = f(x+iy).

For 'real' it is the real part of the output of f(s) = f(x+iy).

For 'imaginary' it is the imaginary part of the output of f(s) = f(x+iy).

For 'argument' or 'phase' it is the argument/phase of the output of f(s) = f(x+iy).
(In other words, if you think of the complex number in polar coordinates, it is the
theta of the output, in the sense of x+iy = rho*(cos(theta) + i*sin(theta)). The
rho is the magnitude.
"""
g(x,y) = f( x + i*y )

if ( (style=='magnitude') or (style=='Magnitude') or (style=='MAGNITUDE') ):
    real_g(x,y) = real_part( g(x,y) )
    imag_g(x,y) = imag_part( g(x,y) )
    answer(x,y) = sqrt( real_g(x,y)^2 + imag_g(x,y)^2 )
elif ( (style=='real') or (style=='Real') or (style=='REAL') ):
    answer(x,y) = real_part( g(x,y) )
elif ( (style=='imaginary') or (style=='Imaginary') or (style=='IMAGINARY')
       or (style=='imag') or (style=='Imag') or (style=='IMAG')):
    answer(x,y) = imag_part( g(x,y) )
elif ( (style=='argument') or (style=='Argument') or (style=='ARGUMENT')
       or (style=='phase') or (style=='Phase') or (style=='PHASE')):
    return newComplexArgumentPlot3d( f, xmin, xmax, ymin, ymax, zmin, zmax, **kwargs )
elif ( (style=='mixed') or (style=='Mixed') or (style=='MIXED')):
    real_g(x,y) = real_part( g(x,y) )
    imag_g(x,y) = imag_part( g(x,y) )
    answer(x,y) = sqrt( real_g(x,y)^2 + imag_g(x,y)^2 )
    T = lambda x,y,z : ( CC( g(x,y) ).argument() + N(pi) ) / (2*N(pi))

    new_color_option = (T, colormaps.gist_rainbow)

    kwargs['color'] = new_color_option
    kwargs['viewer'] = 'tachyon'
```

```
        return new_plot3d( answer(x,y), xmin, xmax, ymin, ymax, zmin, zmax, **kwargs )
    else:
        assert false, "I have no idea what you are asking me for."

    P = new_plot3d( answer(x,y), xmin, xmax, ymin, ymax, zmin, zmax, **kwargs )

    return P
```

Appendix I

Additional Index Entries

Coming soon!