

Lab 3

Plotting With matplotlib and Mayavi

Lab Objective: *Introduce some of the basic plotting functions available in matplotlib and Mayavi.*

matplotlib

matplotlib is one of Python's many plotting libraries. It has an active development community and was designed to resemble MATLAB. Because of its high quality and its helpful user interface toolkits, matplotlib serves as our primary plotting library for 2D graphs in this text.

This section seeks to introduce many of the basic plotting functions we will access using matplotlib, but matplotlib has many more. For further reference and information, visit <http://matplotlib.org>.

Simple Plots

To get started, we import pyplot from matplotlib.

```
from matplotlib import pyplot as plt
```

The basic line plotting function in matplotlib is `plot()`. It takes a set of data points and plots the line that is formed between those points. These plots are pieced together using a *state machine environment*, which means that we can run several different functions that will display or modify the plot we are creating.

To plot and display our function we typically use an outline similar to the following:

```
# Definition of x and y coordinates (typically NumPy arrays or lists).  
plt.plot(x,y)  
plt.show()
```

Note that `x` and `y` must be the same length in order for `plot(x,y)` to work.

When we define our `x` and `y` coordinates, we often use NumPy's `linspace(<start>, <stop>, num=50)` function since it returns evenly spaced values over a given interval.

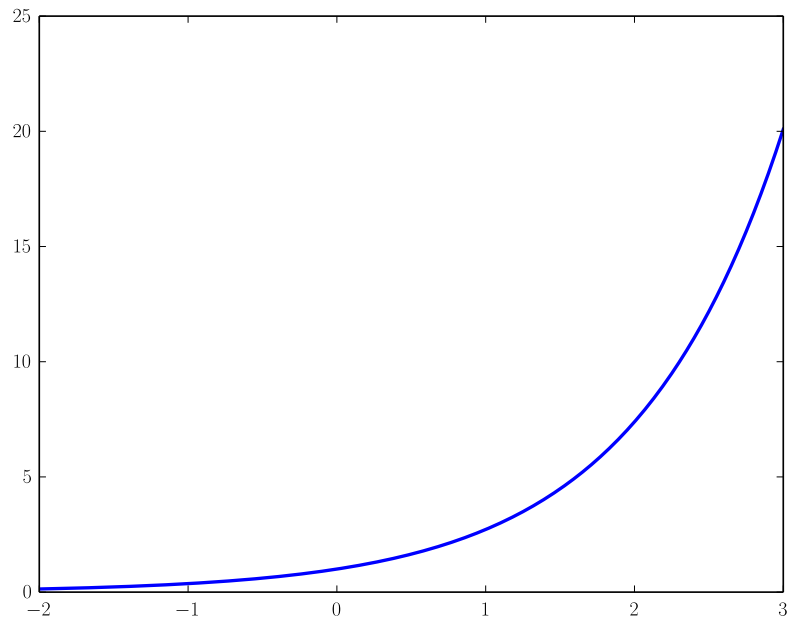


Figure 3.1: A simple plot of e^x .

These values are returned in an `ndarray`. So, after importing `numpy` and `pyplot`, we can generate a basic plot of the function e^x with the following lines of code:

```
# Increase number of samples from default 50 to 501.
x = np.linspace(-2, 3, 501)
y = np.exp(x)
plt.plot(x, y)
plt.show()
```

This should display a plot similar to the one shown in Figure 3.1.

Multiple Plots

We can also use our interface to display multiple plots at once. The following code will plot lines with random values at integers from 1 to 10. Note that NumPy's `random.rand` function creates an array of a specified shape and fills it with random samples from a uniform distribution over the interval $[0,1)$. The given dimensions of the array should be positive and, if no argument is given, a single float is returned.

```
x = np.linspace(1, 10, 10)
y = np.random.rand(10, 10)
plt.plot(x, y[0], x, y[1], x, y[2], x, y[3], x, y[4], x, y[5], x, y[6], x, y[7], ←
        x, y[8], x, y[9])
plt.show()
```

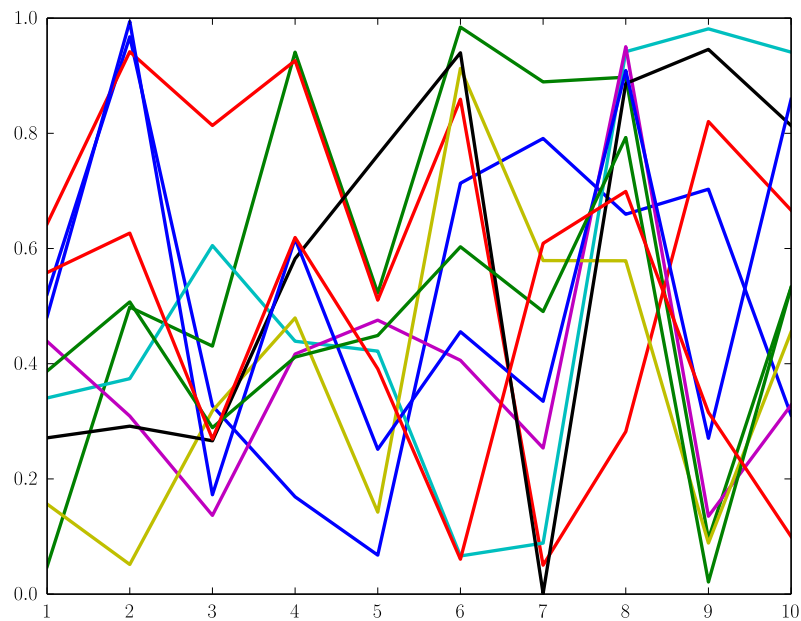


Figure 3.2: A plot of 10 lines with randomly generated y values.

We just used the plot function to plot several different lines at once. Alternatively, we can plot each $(x, y[i])$ pair separately and overlay the plots to produce the same result. We use a loop to efficiently overlay create each plot.

```
x = np.linspace(1, 10, 10)
y = np.random.rand(10, 10)
for n in y:
    plt.plot(x, n)
plt.show()
```

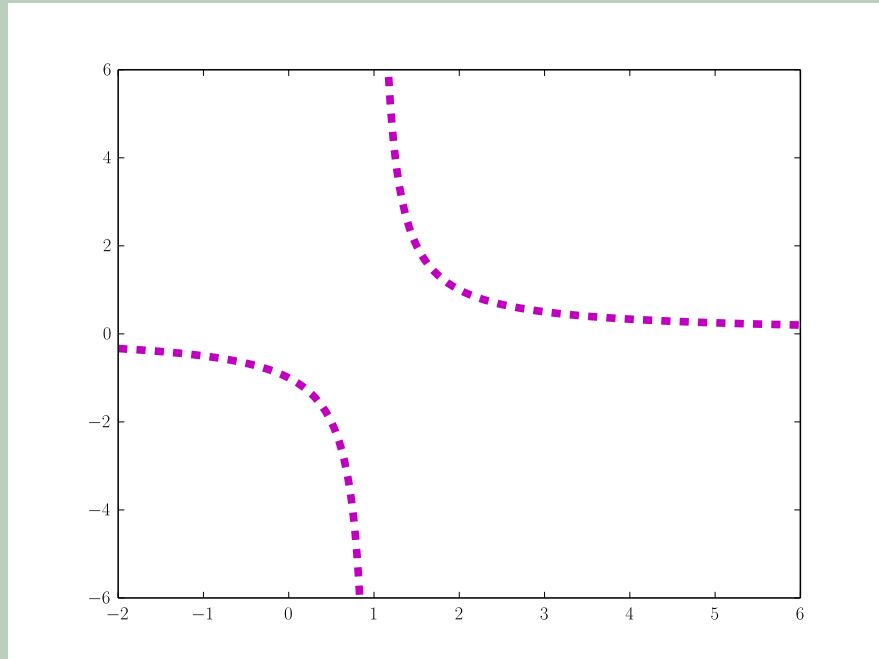
A plot that was generated by this code will be similar to the one shown in Figure 3.2.

Note that a suitable domain and range for your plot is automatically chosen unless you specify otherwise.

Problem 1. Plot the function $\sin(x)$ from 0 to 2π with a red dashed line. Then plot the function $\cos(x)$ on the same domain with a blue dotted line. Implement both with a single call to the `plot()` function. Information on how to do this can be found in Appendix C.

Problem 2. Plot the curve $1/(x - 1)$ from -2 to 6 with a magenta dashed line. Force the plot to only show y values that are between -6 and 6 . Make your line width 5. Once again, more information on how to do this can be found in Appendix C. By default, `plot()` will try to make the graph connected. Correct this so that the graph *appears* to be discontinuous at $x = 1$ (as it should be) by modifying the input values.

Your plot should look like the figure below.



Problem 3. Plot the curve $\sin(x)\frac{1}{x+1}$ from 0 to 10.

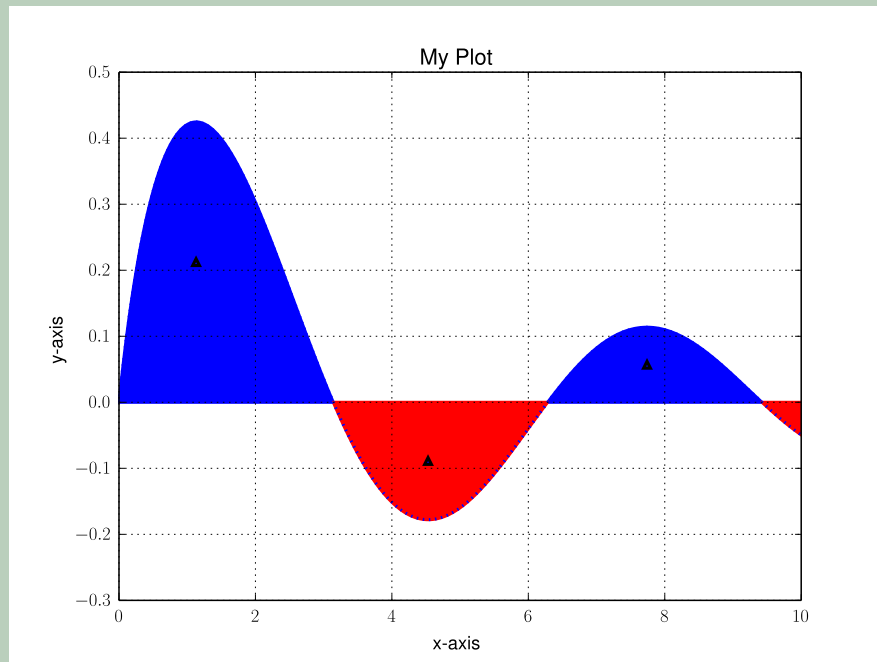
Use blue shading under the curve when it is positive and red when it is negative (you may want to consider using the `fill_between` command. Make the line dotted. Label the x-axis “x-axis”, the y-axis “y-axis”, and the plot “My Plot”. Enable the grid lines.

Finally, use the `scatter` command to include a scatter plot of half of the value of the function at each of its maxima and minima in the range. Display these points as upward-pointing triangles. Don’t forget to make sure the x limits of the plot are still 0 and 10.

Helpful Hint: Since you are working with arrays of discrete values, you will want to find the index values where your x and y values are closest to the actual maxima and minima. As you work, consider the following:

- How would you manually find maxima and minima of a function?
- How could you do something similar with your x and y arrays?

Your plot should look like the figure below.



pcolormesh

Pcolormesh is a function in pyplot which produces a pseudocolor plot of a 2D array. Think of it as a 3D plot that assigns color rather than height to the result C of a function in x and y . Since our *domain* is now two-dimensional, instead of one-dimensional as before, we use NumPy's `meshgrid` function. This function takes two one-dimensional arrays that represent our input values for x and y respectively, and returns a grid of (x, y) coordinates. X represents the x -coordinates of this grid and Y represents the y -coordinates. We then evaluate our function at these points and return the result in a two-dimensional array C , which the `pcolormesh` function uses to assign colors to the points in our xy domain.

We now use the `pcolormesh` function to represent the surface $z = \sin(x) \sin(y)$:

```
n = 401
x = np.linspace(-6, 6, n)
y = np.linspace(-6, 6, n)
# Returns a coordinate matrix given coordinate vectors.
X, Y = np.meshgrid(x, y)
C = np.sin(X) * np.sin(Y)
plt.pcolormesh(X, Y, C)
plt.show()
```

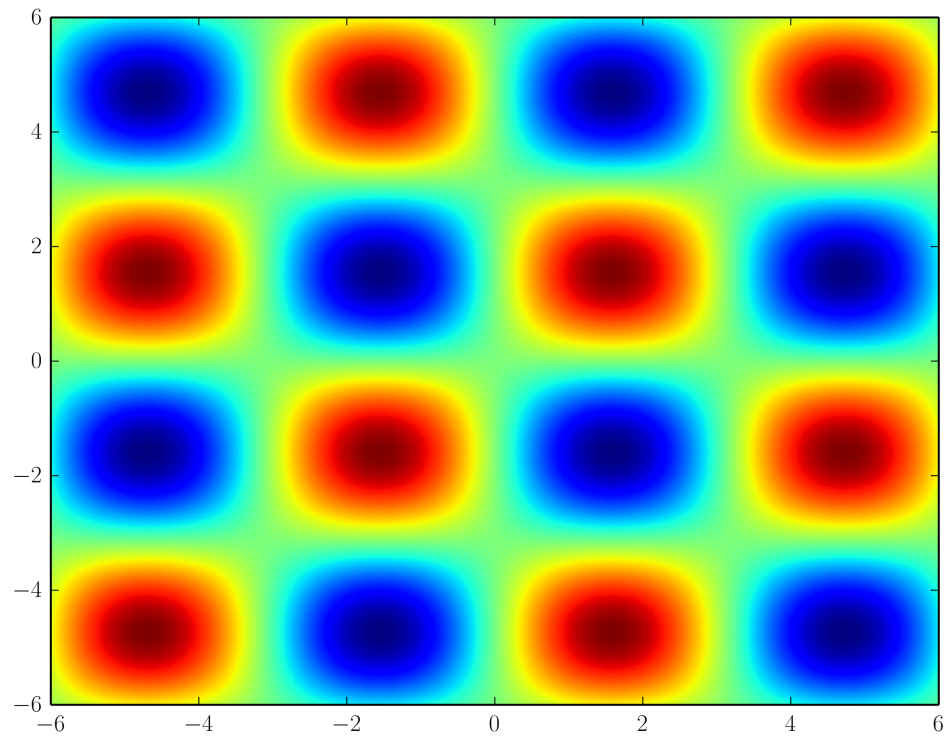
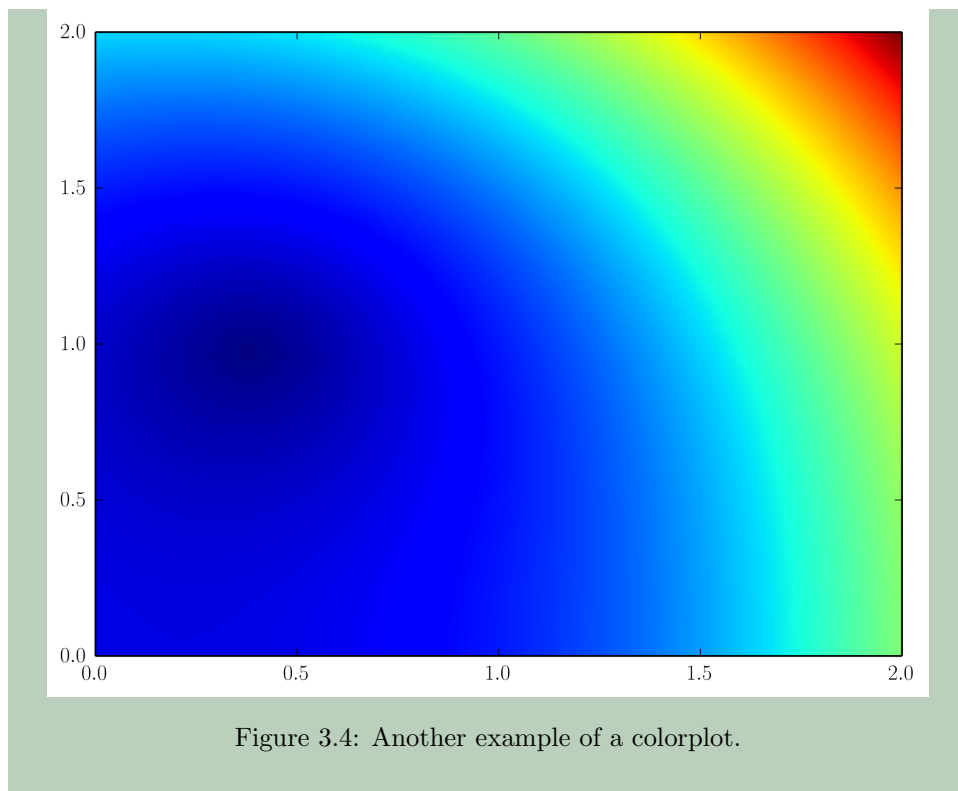


Figure 3.3: Color plot of $\sin(x) \times \sin(y)$.

This plot is shown in Figure 3.3

Problem 4. Use `plt.pcolormesh` to plot the absolute value of the function $x^3 + 2x^2 - x + 3$ on the complex plane with $0 \leq x \leq 2$ and $0 \leq y \leq 2$. *Helpful Hint:* First create your domain arrays, then convert these to a single array of complex variables to evaluate the function.

Your plot should look like Figure 3.4.



subplots

We can plot multiple different images within the same figure using the `plt.subplot` command. It takes three arguments: the total number of rows for the figure, the total number of columns for the figure, and the index of the current subplot. This index starts at 1 and increments across rows first. Preface the code for each subplot with the following command:

```
plt.subplot(numrows, numcols, fignum)
```

If all the argument values are less than 10, commas and spaces can be omitted.

The following example shows plots of $\sin x$ and $\cos x$ on two different axes within the same figure. Figure 3.5 is the output from this code.

```
x = np.linspace(-np.pi, np.pi, 400)
y1 = np.sin(x)
y2 = np.cos(x)
# Commas and spaces are omitted between arguments.
plt.subplot(211)
plt.plot(x, y1)
plt.subplot(212)
plt.plot(x, y2)
plt.show()
```

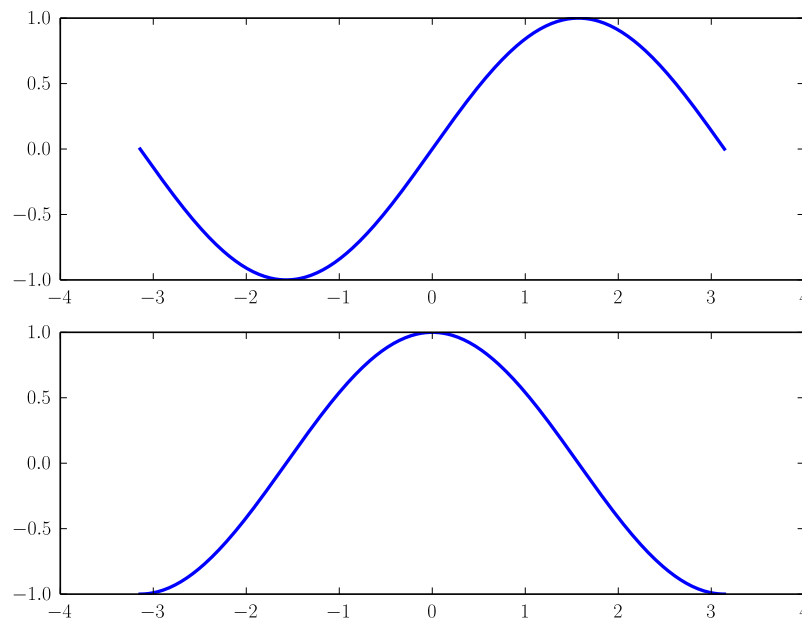


Figure 3.5: An example of the use of subplots in matplotlib.

Titles and Labels

We can add titles to each subplot using the `plt.title` function. The `plt.suptitle` function allows us to give the entire figure a title as well.

Problem 5. Make a plot with 4 subplots. In the subplots place graphs of e^x , $\sin(x)$, $\cos(x)$, and x^2 . Plot each graph over the interval $(-\pi, \pi)$. Title each graph accordingly and title the entire figure “My Different Plots.”

Mayavi

matplotlib is designed primarily for 2D plotting. Although matplotlib can also create basic 3D plots, it is better to create 3D plots using a library that was specifically designed to do so: Mayavi. Not only is Mayavi designed primarily for 3D plotting, it further has the benefits of easy integration with Python scientific libraries and a user interface that allows for interaction with all data and objects.

We encourage you to visit the formal documentation at <http://docs.enthought.com/mayavi/mayavi/> for more information.

Within Mayavi, we will be using the `mlab` API which is imported like so:

```
from mayavi import mlab
```

`mlab` has many plotting functions, which are detailed at http://docs.enthought.com/mayavi/mayavi/auto/mlab_helper_functions.html. All these functions can be “tested” (which provides an example figure for the function) with the command:

```
mlab.test_<plotting function>()
mlab.show()
```

Note that pressing tab after typing `mlab.test_` will provide a list of available commands and that, like `matplotlib`, we use the `show()` command to actually view our plot.

For example, to test the `fancy_mesh` function, we run the following:

```
mlab.test_fancy_mesh()
mlab.show()
```

In this lab, we will focus our attention on introducing the `mesh`, `plot3d`, and `points3d` functions.

Like `matplotlib`, `mlab` takes a set of data points and plots it. However, instead of passing just `x` and `y` coordinates as we did in `matplotlib`, we also send `z` coordinates to produce a 3D plot.

mesh

`mesh` plots a surface using grid-spaced data. It expects three 2D NumPy arrays (i.e., a grid of `(x,y,z)` coordinates). As with `pcolormesh` in `matplotlib`, it is important that all of the given arrays have the same shape. The following illustrates a hyperbolic paraboloid, using the colormap `RdYlGn`.

```
import numpy as np
from mayavi import mlab

x = np.linspace(-4,4,300)
y = np.linspace(-4,4,300)
X, Y = np.meshgrid(x,y)

Z = X**2/4 - Y**2/4

mlab.mesh(X, Y, Z, colormap='RdYlGn')
mlab.show()
```

This will produce Figure 3.6

plot3d

`plot3d` is used to draw lines between points. It expects three one-dimensional NumPy arrays to provide the position of the line.

The following illustrates a simple floral design.

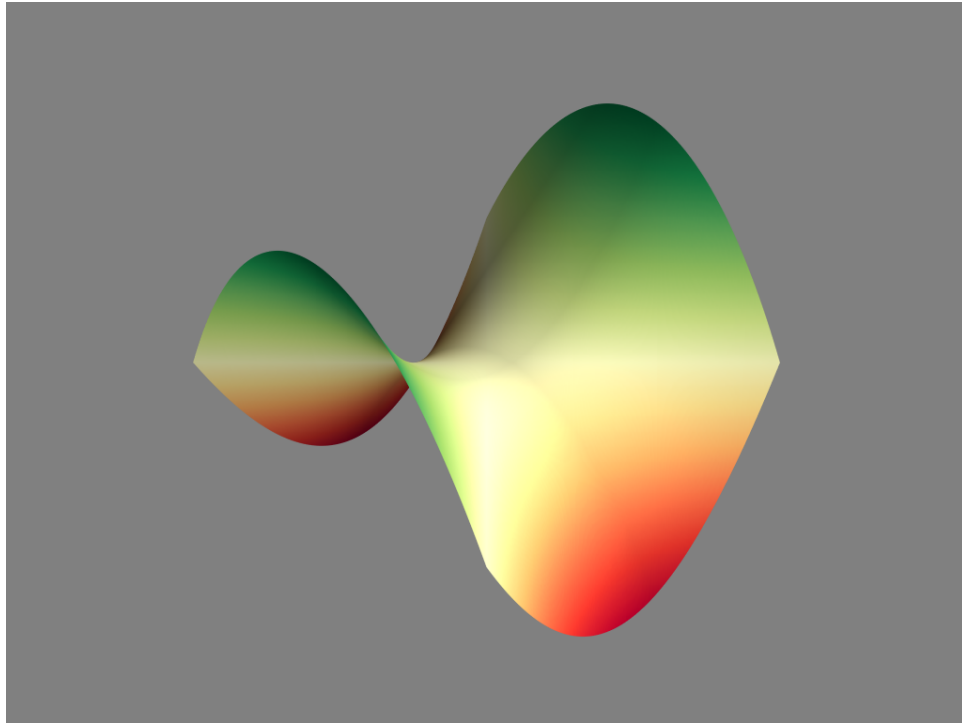


Figure 3.6: A hyperbolic paraboloid

```

num = np.pi/1000
pts = np.arange(0, 2*np.pi + num, num)
x = np.cos(pts) * (1 + np.cos(pts*6))
y = np.sin(pts) * (1 + np.cos(pts*6))
z = np.sin(pts*6/11)
mlab.plot3d(x, y, z)
mlab.show()

```

This will produce Figure 3.7

points3d

`points3d` is similar to `plot3d`, except that it plots glyphs (the three-dimensional analogue of points) at the positions of the supplied data, and does not connect them like `plot3d` does. It takes the same arguments, three 1D NumPy arrays that form a line given by a list of (x,y,z) coordinates. For this example, we will also use a keyword argument, `s`, which provides an associated scalar value for each point given. This modifies the size and color of the glyphs. We also use a scale factor to adjust the size of the points so that all of them are visible.

The following illustrates a simple heart - given the right perspective.

```

pts = np.linspace(0, 4 * np.pi, 30)
x = np.sin(2 * pts)
y = np.cos(pts)

```

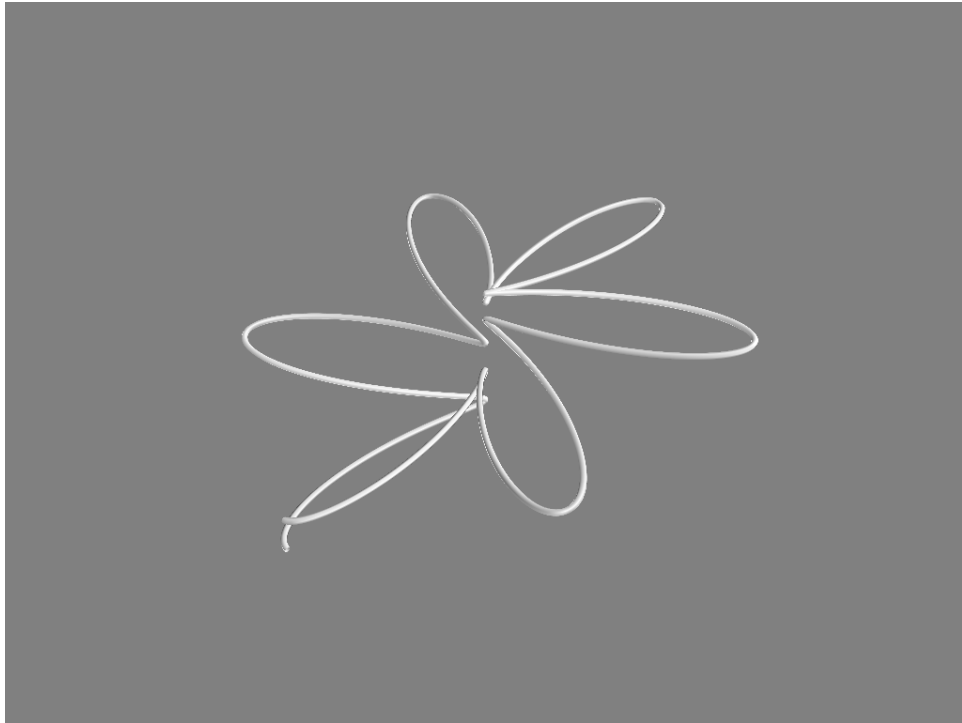


Figure 3.7: A simple floral.

```
z = np.cos(2 * pts)
s = 2*np.sin(pts)
mlab.points3d(x, y, z, s, scale_factor=.15)
mlab.show()
```

This will produce Figure 3.8.

Changing Looks

The color of your object can be explicitly defined using the `color` keyword argument. It is specified as a triplet (red, green, blue) of floating points ranging from 0 to 1. For example, (1.0, 1.0, 1.0) is white.

If, however, you would prefer to vary the colors across your visualization, we recommend using a colormap. For example, in Figure 3.6 we used the colormap `RdYlGn` to make our hyperbolic paraboloid a blend from red to cream to green. Visit http://docs.enthought.com/mayavi/mayavi/mlab_changing_object_looks.html for a list of all previously established colormaps.

Useful Features

Another useful feature in `mayavi` is the record feature. This can be found by clicking the pipeline view in the top left corner of a plotting window. In the pipeline view

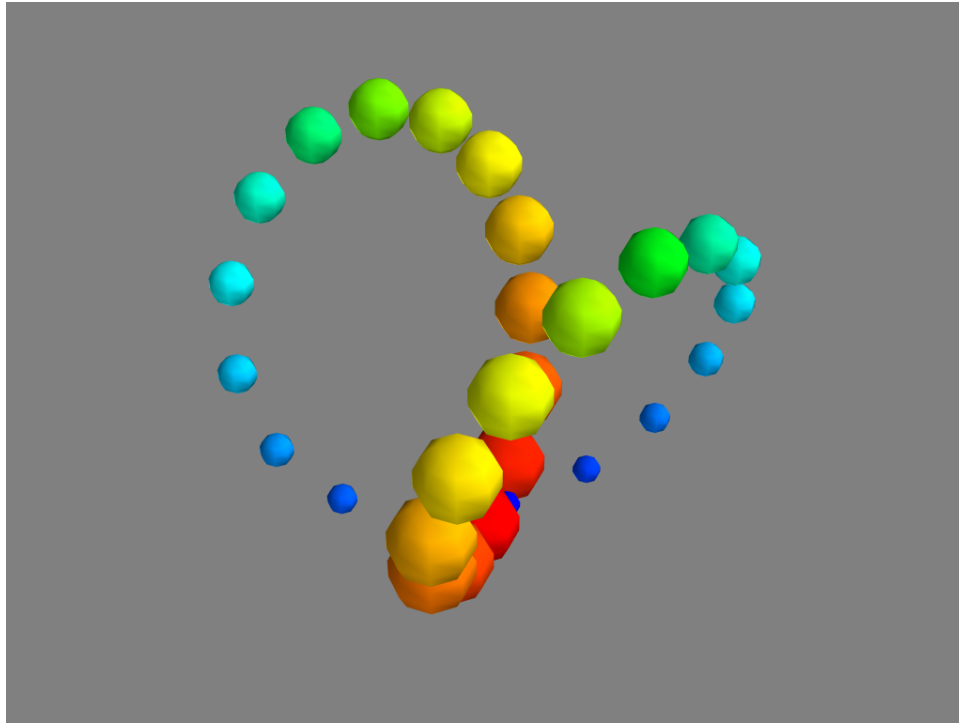


Figure 3.8: A simple heart.

there is a round, red button. Clicking it opens a recorder that keeps track of all the changes made interactively to the visualization, using valid lines of Python code.

Problem 6. Provided Grand Canyon topological radar data from NASA, do the following.

- Reshape the data to be 3601x3601.
- Cast the data type as `float32`.
- Slice the data, taking the first 1000 rows and columns 900-1900.
- There is some missing data, so set the minimum of your data equal to the minimum of the the positive data points.
- Preset the figure using the following commands: `mlab.figure(size=(400,320), bgcolor = (.16, .28, .46))`
- Now plot with `mlab.surf`, using the colormap `gist_earth`, with a `warp_scale = .2`, `vmin=1200`, and `vmax=1610`.
- Take a smaller view of the canyon using `mlab.view(-5.9, 83, 570, [5.3, 20, 238])`.

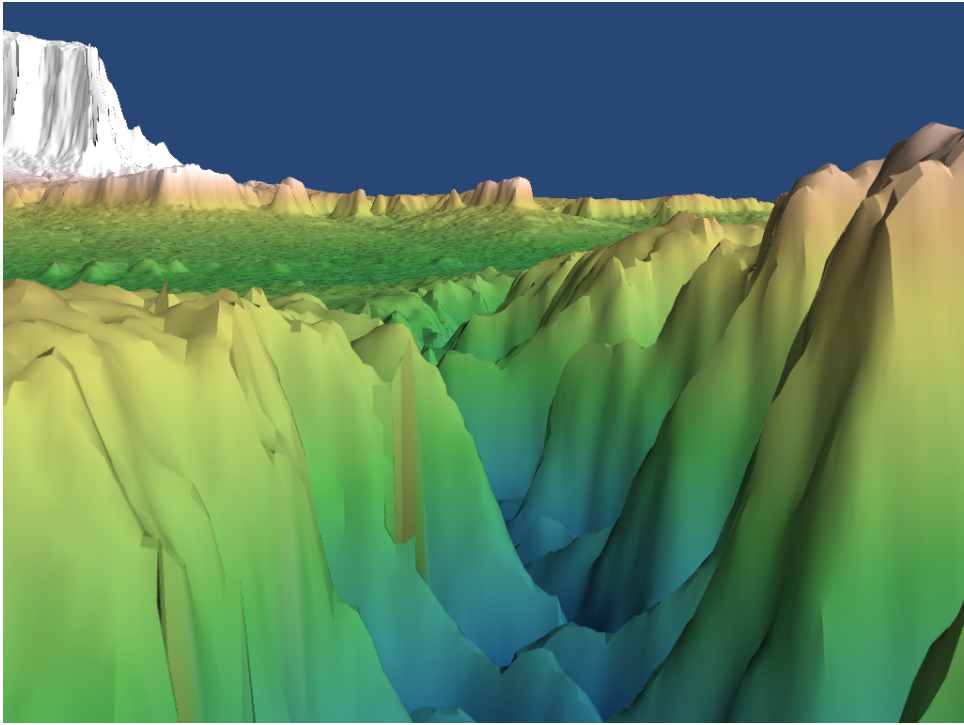


Figure 3.9: A view of the Grand Canyon

This should produce Figure 3.9