Lab 1

Plotting With matplotlib and Mayavi

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

2-D plotting with matplotlib

The Python library `matplotlib` will be our primary tool for creating 2-D graphs in this text. This lab introduces the basic features of `matplotlib`; for more information, visit the documentation at `http://matplotlib.org`.

To begin, import `pyplot` from `matplotlib`.

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

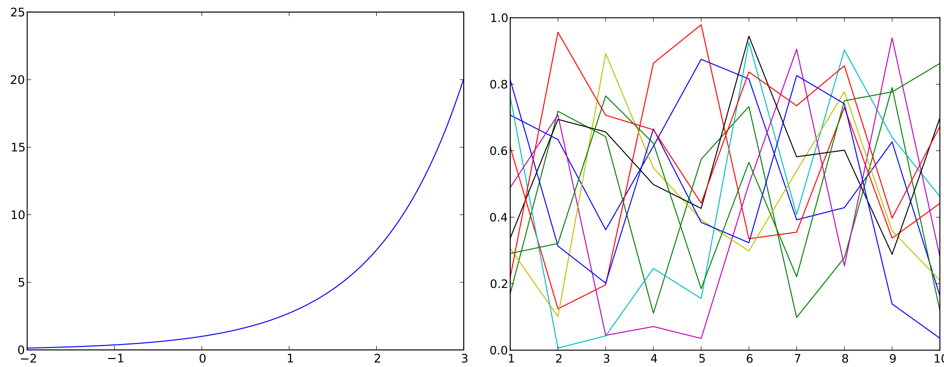
Line plots

The function call `plt.plot(x, y)` takes two 1-D NumPy arrays, `x` and `y`, and plots the points `(x[i], y[i])`, connecting them with straight lines.

For example, this code plots $y = e^x$ on the interval $[-2, 3]$. The output is in Figure 1.1a.

Note

If you are executing these matplotlib commands in an IPython shell, executing the `plt.show` method will open a new window with the plot. If you are using IPython Notebook, you have the option to display the plots within your notebook. You may opt into this feature by running `%matplotlib inline` or `%matplotlib notebook` in your IPython Notebook. The `inline` option shows the plot, whereas the `notebook` option shows the plot and provides controls to interact with the plot. Additionally, when using this option, the plot is displayed after running the `plt.plot` command; the `plt.show` command is not necessary.



(a) A graph of e^x.

(b) A graph of ten lines with randomly generated y values.

Figure 1.1: Plots created with `plt.plot()`.

```
import numpy as np
# Create an array of 501 evenly spaced points in [-2,3].
x = np.linspace(-2, 3, 501)
y = np.exp(x)
plt.plot(x, y)
plt.show()
```

All calls to `plt` functions will modify the same figure until you call `plt.show()`. The function `plt.show()` displays the current figure and resets the system, so that the next call to a `plt` function modifies a new figure.

We can take advantage of this system to plot multiple lines on the same axes. For example, the following code plots ten lines, each with random values at integers from 1 to 10. The output is in Figure 1.1b.

```
x = np.linspace(1, 10, 10)

# Create a 10x10 array of uniformly distributed values in [0,1).
y = np.random.rand(10, 10)

# Plot each row of y
for row in y:
    plt.plot(x, row)
plt.show()
```

Alternatively, we can produce the same graph with a single call to `plt.plot()`.

```
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()
```

Problem 1. Plot the curve $1/(x-1)$ on [-2,6]. Do the following to your plot:

1. The function `plt.plot()` will make the curve look continuous at $x = 1$. Plot the two sides of the curve separately (still with a single call to `plt.plot()`) so that the graph looks discontinuous at $x = 1$.
2. Plot the curve with a line that is magenta, dashed, and thick (see Appendix **??**).
3. Change the range of the y-axis to be $[-6, 6]$.

Your final plot should look like Figure 1.2.

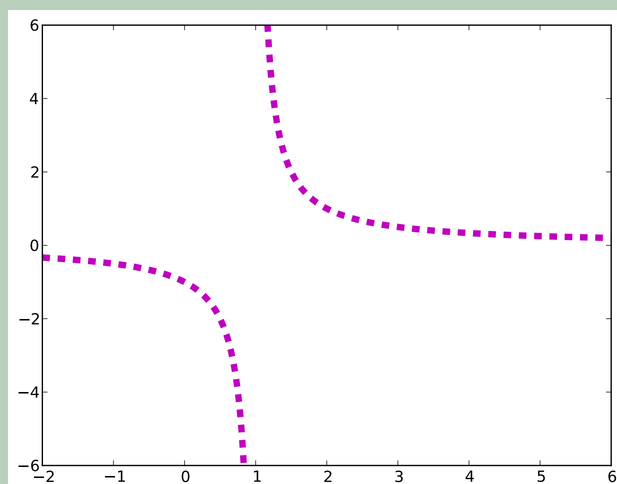


Figure 1.2: Correct output for Problem 1.

Heatmaps

A function from $\mathbb{R}^2$ to $\mathbb{R}$ is usually plotted as a surface in $\mathbb{R}^3$. A *heatmap* visualizes such a function in only two dimensions by assigning the output of the function to a color (instead of a height). For example, Figure 1.4 uses a heatmap to graph $f(x, y) = \sin(x)\sin(y)$ on $[-6, 6] \times [-6, 6]$.

The plot in Figure 1.4 was created with the function `plt.pcolormesh()`. To draw Figure 1.4, we must first create a grid of points at which to evaluate f. We do this with the function `np.meshgrid()`, which is explained in Figure 1.3.

```
# Create an XY meshgrid
x = np.linspace(-6, 6, 401)
y = np.linspace(-6, 6, 401)
X, Y = np.meshgrid(x, y)
```

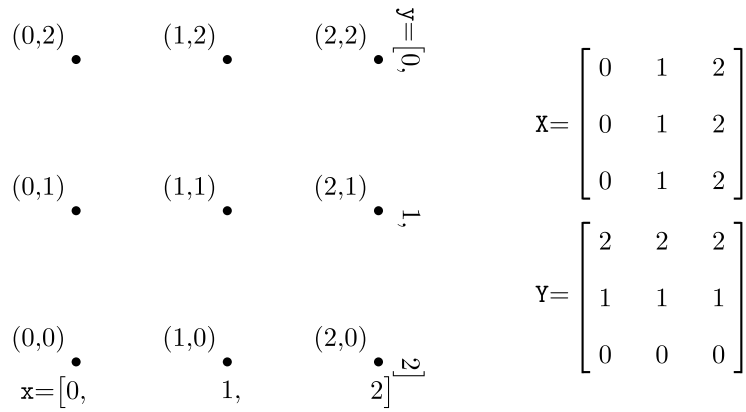


Figure 1.3: This figure illustrates the function call `np.meshgrid(x, y)`, which returns the arrays `X` and `Y`. The returned arrays give the x- and y-coordinates of the points in the grid formed by `x` and `y`.

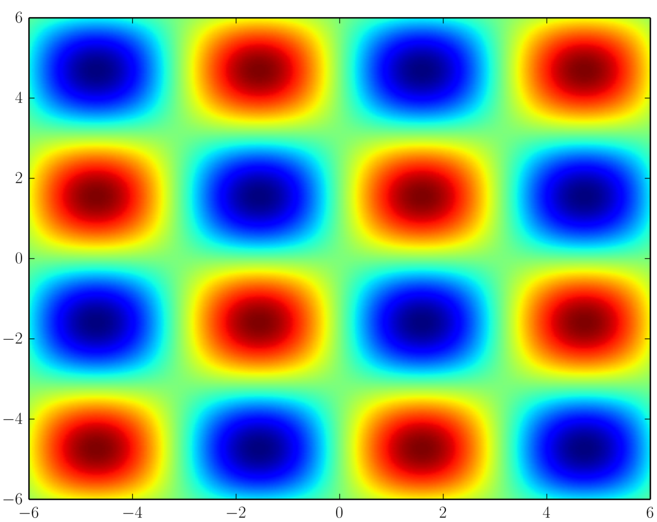


Figure 1.4: A heatmap of $f(x, y) = \sin(x) \times \sin(y)$ drawn by `plt.pcolormesh()`.

The arrays X and Y satisfy `(X[i,j], Y[i,j]) = (x[i],y[j])`.

Now we can evaluate $f(x, y)$ at each point in the grid and plot the result.

```
f = np.sin(X) * np.sin(Y)
plt.pcolormesh(X, Y, f)
plt.colorbar()    # Show scale
plt.show()
```

This plot is shown in Figure 1.4

Problem 2.

1. Plot the function $f(x,y) = \sin(x)\sin(y)/(xy)$ on $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$. Include the scale bar in your plot.
2. Change the color scheme of your plot with the keyword argument `cmap='seismic'` in the call to `plt.pcolormesh()`. You can see a list of all possible color schemes at `http://matplotlib.org/examples/color/colormaps_reference.html`.
3. Change the limits on the x- and y-axes so that the plot is only over the domain $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$.
4. Fix the aspect ratio of your plot so that it is a square using the line `plt.gca().set_aspect('equal')`.

Your finished plot should look like Figure 1.5.

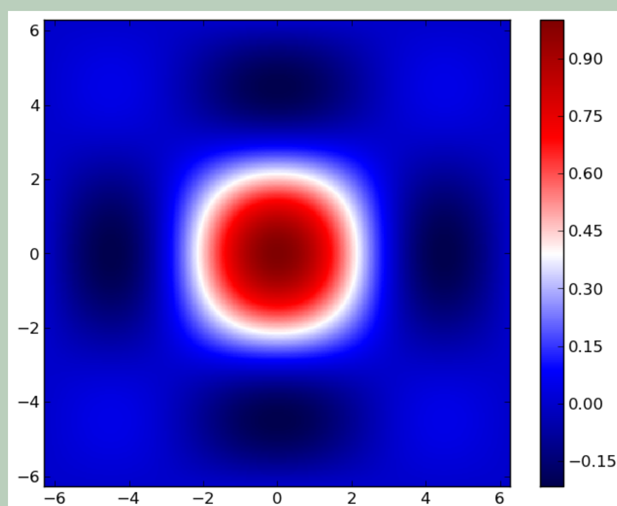


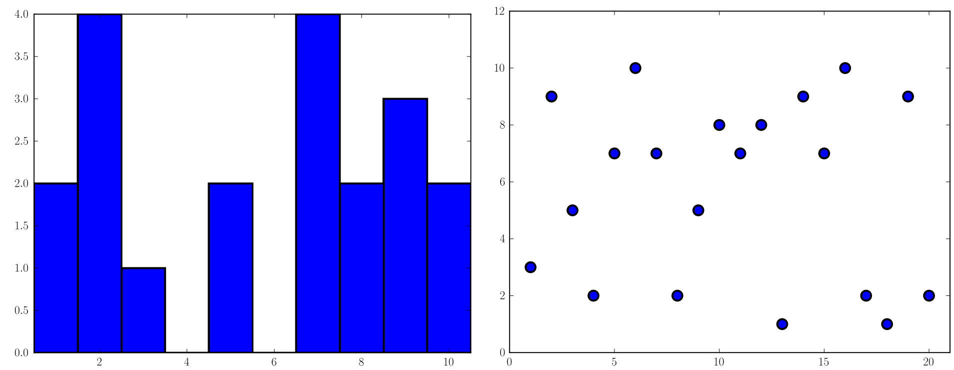
Figure 1.5: Correct output for Problem 2.

Other plots

A *histogram* is a way to visualize a 1-D data set, or list of values. A histogram is created by dividing up the range of the values into a finite number of intervals, or *bins*. Then, the number of values in each bin is added up. Graphically, these totals are represented by bars whose length is equal to the number of values in the bin.

For example, suppose we randomly choose 20 integers between 1 and 10. This code creates a histogram depicting how many of each number we chose. Its output is in Figure 1.6a.

```
x = np.random.randint(1, 11, 20)
```



(a) A histogram.

(b) A scatter plot.

Figure 1.6: A histogram produced with `plt.hist()` and a scatter plot produced with `plt.scatter()`.

```
plt.hist(x, bins=10, range=[.5, 10.5])
plt.show()
```

In this example our data set `x` consists of only integer values. We created 10 bins in the range [.5, 10.5] so that each bin contains exactly one integer.

The function `plt.hist()` also returns some arrays, the first of which is a list of the total number of values in each bin.

We could also use a *scatter plot* to visualize the random integers `x` that we generated in the previous example. The `matplotlib` function call `plt.scatter(x,y)` draws a scatter plot from two 1-D arrays `x` and `y` by plotting the points `(x[i], y[i])`. As an example, the code below produces Figure 1.6b.

```
t = np.linspace(1,20,20)
# The argument 's' specifies the marker size.
plt.scatter(t, x, s=100)
plt.show()
```

Subplots

Subplots are nonoverlapping plots arranged in a grid within a single figure (see Figure 1.7). In `matplotlib`, specify which subplot you wish to modify with command `plt.subplot(numrows, numcols, fignum)`. Here, `numrows` is the number of rows of subplots in the figure, `numcols` is the number of columns of subplots in the figure, and `fignum` is the index of the subplot you wish to modify. This index starts at 1 and increments across rows first.

The following code draws Figure 1.7.

```
x = np.linspace(-np.pi, np.pi, 400)
y1 = np.sin(x)
y2 = np.cos(x)

# Draw the first subplot.
```

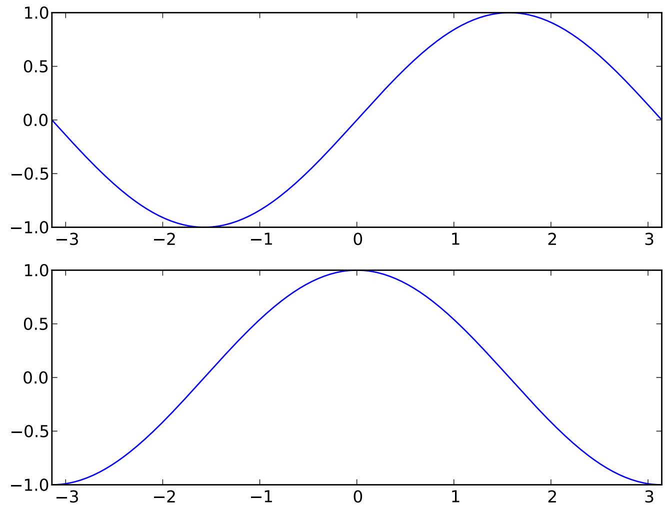


Figure 1.7: The graphs of $\sin(x)$ and $\cos(x)$ as subplots in a single figure.

```
plt.subplot(2, 1, 1)
plt.plot(x, y1)

# Draw the second subplot.
plt.subplot(2, 1, 2)
plt.plot(x, y2)

plt.show()
```

Problem 3. Follow these steps to generate a plot similar to that in Figure 1.8:

1. Generate 50 random numbers in the interval $[0, 1)$ with the function `np.random.rand()`.
2. Create a plot with two subplots. In the first subplot, draw a histogram of your data with 5 equally-sized bins.
3. In the second subplot,
   (a) Draw a scatter plot of your data. Use the integers 1-50 for the x-coordinates and use your data for the y-coordinates.
   (b) Plot a red horizontal line whose height is equal to the mean of your data.

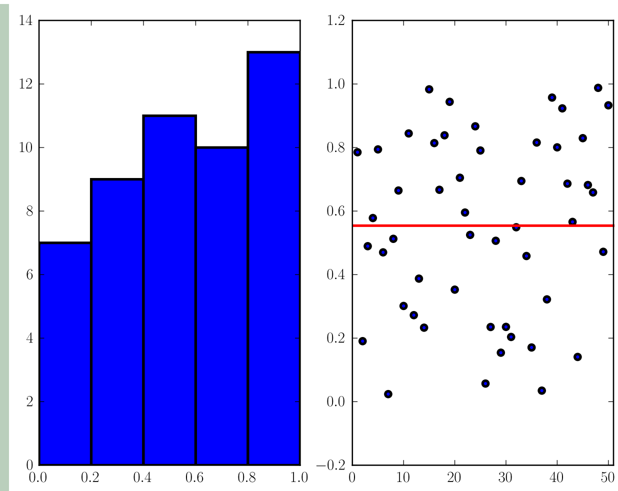


Figure 1.8: Correct output for Problem 3.

3-D plotting with Mayavi

Although `matplotlib` is capable of creating 3-D plots, Mayavi does it better. We will use Mayavi for all 3-D plots in these labs. For information beyond what is found in this tutorial, see `http://docs.enthought.com/mayavi/mayavi/`. To get started, import `mlab` from `mayavi`.

```
from mayavi import mlab
```

Note

If you do not have the `Mayavi` package installed on your system, you may download it by running the following commands from the command line:

```
$ conda install mayavi
```

For more information regarding installing Python packages, see Appendix **??**.

The module `mlab` has many plotting functions. We will introduce a few in this lab; for more see `http://docs.enthought.com/mayavi/mayavi/auto/mlab_helper_functions.html`. You can also browse examples at `http://docs.enthought.com/mayavi/mayavi/auto/examples.html`.

Lines

The function call `mlab.plot3d(x,y,z)` plots the points `(x[i], y[i], z[i])` and connects them with straight lines. The code below produces the flower in Figure 1.9

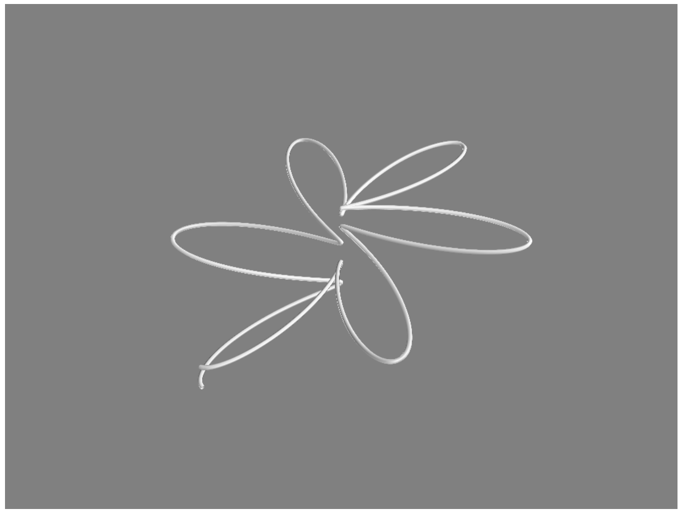


Figure 1.9: Sample output of `mlab.plot3d()`.

```
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()
```

Points

The function call `mlab.points3d(x,y,z)` plots the points `(x[i], y[i], z[i])`, but does not connect them. In the code below, the optional input array `s` defines a scalar for each point that modifies the color and size of the point. The output is in Figure 1.10.

```
pts = np.linspace(0, 4 * np.pi, 30)
x = np.sin(2 * pts)
y = np.cos(pts)
z = np.cos(2 * pts)
s = 2+np.sin(pts)
# Adjust the keyword argument 'scale_factor' so all points are visible.
mlab.points3d(x, y, z, s, scale_factor=.15)
mlab.show()
```

Surfaces

You can draw a surface in mayavi with `mlab.surf()`. This function accepts three arrays, `X`, `Y`, and `Z`, where `X` and `Y` determine a grid of points similar to the output of `np.meshgrid()`. The array `Z` gives the height of the surface at each of these points.

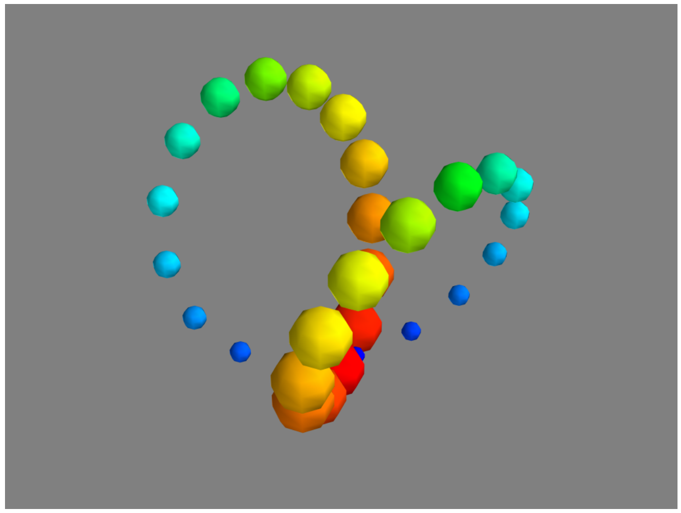


Figure 1.10: Sample output of `mlab.points3d()`.

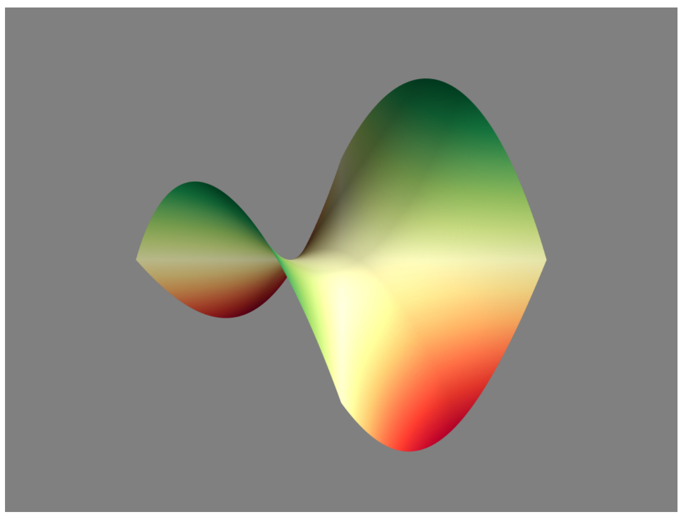


Figure 1.11: Sample output of `mlab.surf()`.

The output of `np.meshgrid()` is the transpose of what `mlab.surf()` expects. To avoid confusion, use the function `np.mgrid()` instead. This function uses the slicing syntax `[start:stop:step]` The code below produces the hyperbolic paraboloid in Figure 1.11.

```
X, Y = np.mgrid[-4:4:0.025, -4:4:0.025]
Z = X**2/4-Y**2/4
mlab.surf(X, Y, Z, colormap='RdYlGn')
mlab.show()
```

The plotting functions in Mayavi allow you to specify the color of a plot, either as a solid color or with a varying colormap. For example, the plot in Figure 1.11

uses the colormap `'RdYlGn'`. For a list of all colormaps in Mayavi, see `http://docs.enthought.com/mayavi/mayavi/mlab_changing_object_looks.html`.

Problem 4. Plot the function $z = \sin(10(x^2 + y^2))/10$ on $[-1, 1] \times [-1, 1]$ using Mayavi.