5 Introduction to Matplotlib

Lab Objective: *Matplotlib is the most commonly used data visualization library in Python. Being able to visualize data helps to determine patterns and communicate results and is a key component of applied and computational mathematics. In this lab we introduce techniques for visualizing data in 1, 2, and 3 dimensions. The plotting techniques presented here will be used in the remainder of the labs in the manual.*

Line Plots

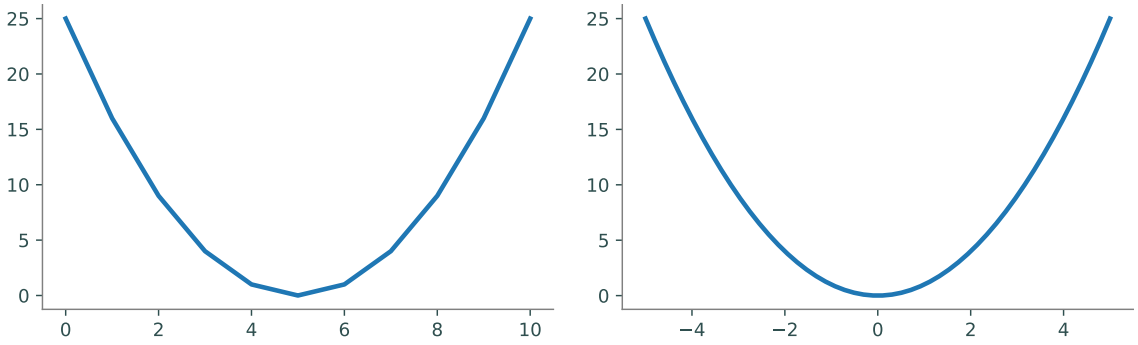
Raw numerical data is rarely helpful unless it can be visualized. The quickest way to visualize a simple 1-dimensional array is with a *line plot*. The following code creates an array of outputs of the function $f(x) = x^2$, then visualizes the array using the `matplotlib` module[1] [Hun07].

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt

>>> y = np.arange(-5,6)**2
>>> y
array([25, 16,  9,  4,  1,  0,  1,  4,  9, 16, 25])

# Visualize the plot.
>>> plt.plot(y)                     # Draw the line plot.
[<matplotlib.lines.Line2D object at 0x1084762d0>]
>>> plt.show()                      # Reveal the resulting plot.
```

The result is shown in Figure 5.1a. Just as `np` is a standard alias for NumPy, `plt` is a standard alias for `matplotlib.pyplot` in the Python community.

The call `plt.plot(y)` creates a figure and draws straight lines connecting the entries of `y` relative to the y-axis. The x-axis is (by default) the index of the array, which in this case is the integers from 0 to 10. Calling `plt.show()` then displays the figure.

[1] Like NumPy, Matplotlib is *not* part of the Python standard library, but it is included in most Python distributions. See `https://matplotlib.org/` for the complete Matplotlib documentation.



(a) `plt.plot(y)` uses the indices of the array for the x-axis.

(b) `plt.plot(x,y)` specifies both the domain and the range.

Figure 5.1: Plots of $f(x) = x^2$ over the interval $[-5, 5]$.

Problem 1. NumPy's `random` module has tools for sampling from probability distributions. For instance, `np.random.normal()` draws samples from the normal (Gaussian) distribution. The `size` parameter specifies the shape of the resulting array.

```
>>> np.random.normal(size=(2,3))    # Get a 2x3 array of samples.
array([[ 1.65896515, -0.43236783, -0.99390897],
       [-0.35753688, -0.76738306,  1.29683025]])
```

Write a function that accepts an integer n as input.

1. Use `np.random.normal()` to create an $n \times n$ array of values randomly sampled from the standard normal distribution.
2. Compute the mean of each row of the array.
   (Hint: Use `np.mean()` and specify the `axis` keyword argument.)
3. Return the variance of these means.
   (Hint: Use `np.var()` to calculate the variance).

Define another function that creates an array of the results of the first function with inputs $n = 100,\ 200,\ \ldots,\ 1000$. Plot (and show) the resulting array.

Specifying a Domain

An obvious problem with Figure 5.1a is that the x-axis does not correspond correctly to the y-axis for the function $f(x) = x^2$ that is being drawn. To correct this, define an array `x` for the domain, then use it to calculate the image `y = f(x)`. The command `plt.plot(x,y)` plots `x` against `y` by drawing a line between the consecutive points `(x[i], y[i])`.

Another problem with Figure 5.1a is its poor resolution: the curve is visibly bumpy, especially near the bottom of the curve. NumPy's `linspace()` function makes it easy to get a higher-resolution domain. Recall that `np.arange()` returns an array of evenly-spaced values in a given interval, where

the **spacing** between the entries is specified. In contrast, `np.linspace()` creates an array of evenly-spaced values in a given interval where the **number of elements** is specified.

```
# Get 4 evenly-spaced values between 0 and 32 (including endpoints).
>>> np.linspace(0, 32, 4)
array([  0.        ,  10.66666667,  21.33333333,  32.        ])

# Get 50 evenly-spaced values from -5 to 5 (including endpoints).
>>> x = np.linspace(-5, 5, 50)
>>> y = x**2                          # Calculate the image of f(x) = x**2.
>>> plt.plot(x, y)
>>> plt.show()
```

The resulting plot is shown in Figure 5.1b. This time, the x-axis correctly matches up with the y-axis. The resolution is also much better because `x` and `y` have 50 entries each instead of only 10.

Subsequent calls to `plt.plot()` modify the same figure until `plt.show()` is executed, which displays the current figure and resets the system. This behavior can be altered by specifying separate figures or axes, which we will discuss shortly.

Note

Plotting can seem a little mystical because the actual plot doesn't appear until `plt.show()` is executed. Matplotlib's *interactive mode* allows the user to see the plot be constructed one piece at a time. Use `plt.ion()` to turn interactive mode on and `plt.ioff()` to turn it off. This is very useful for quick experimentation. Try executing the following commands in IPython:

```
In [1]: import numpy as np
In [2]: from matplotlib import pyplot as plt

# Turn interactive mode on and make some plots.
In [3]: plt.ion()
In [4]: x = np.linspace(1, 4, 100)
In [5]: plt.plot(x, np.log(x))
In [6]: plt.plot(x, np.exp(x))

# Clear the figure, then turn interactive mode off.
In [7]: plt.clf()
In [8]: plt.ioff()
```

Use interactive mode **only** with IPython. Using interactive mode in a non-interactive setting may freeze the window or cause other problems.

Problem 2. Write a function that plots the functions $\sin(x)$, $\cos(x)$, and $\arctan(x)$ on the domain $[-2\pi, 2\pi]$ (use `np.pi` for π). Make sure the domain is refined enough to produce a figure with good resolution.

Plot Customization

`plt.plot()` receives several keyword arguments for customizing the drawing. For example, the color and style of the line are specified by the following string arguments.

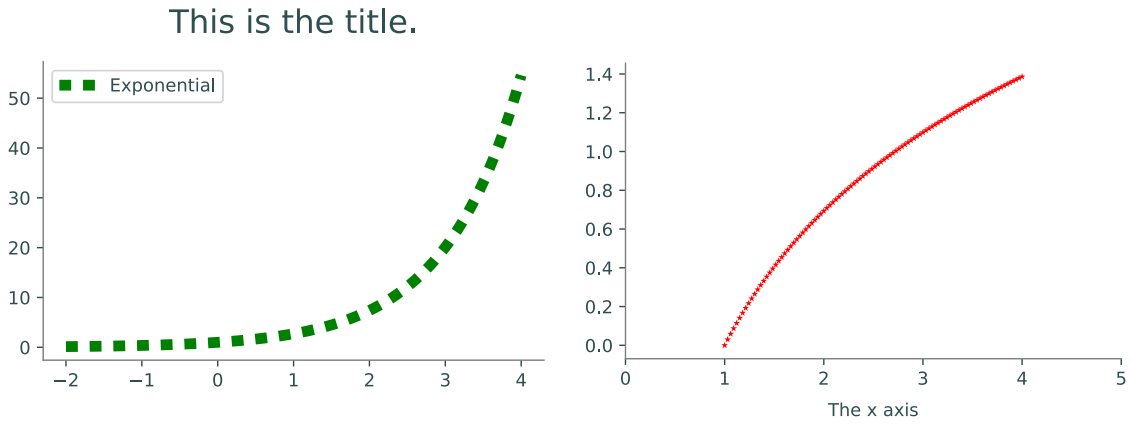
Key	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'k'`	black

Key	Style
`'-'`	solid line
`'--'`	dashed line
`'-.'`	dash-dot line
`':'`	dotted line
`'o'`	circle marker

Specify one or both of these string codes as the third argument to `plt.plot()` to change from the default color and style. Other `plt` functions further customize a figure.

Function	Description
`legend()`	Place a legend in the plot
`title()`	Add a title to the plot
`xlim()` / `ylim()`	Set the limits of the x- or y-axis
`xlabel()` / `ylabel()`	Add a label to the x- or y-axis

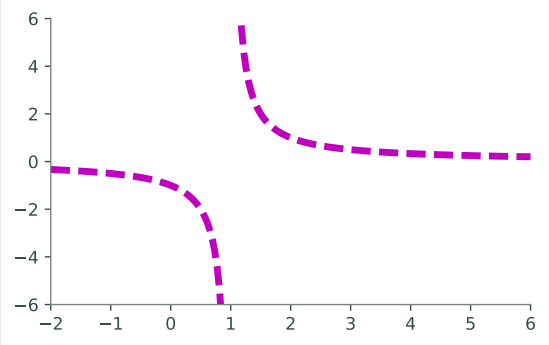
```
>>> x1 = np.linspace(-2, 4, 100)
>>> plt.plot(x1, np.exp(x1), 'g:', linewidth=6, label="Exponential")
>>> plt.title("This is the title.", fontsize=18)
>>> plt.legend(loc="upper left")    # plt.legend() uses the 'label' argument of
>>> plt.show()                      # plt.plot() to create a legend.

>>> x2 = np.linspace(1, 4, 100)
>>> plt.plot(x2, np.log(x2), 'r*', markersize=4)
>>> plt.xlim(0, 5)                  # Set the visible limits of the x axis.
>>> plt.xlabel("The x axis")        # Give the x axis a label.
>>> plt.show()
```



Problem 3. Write a function to plot the curve $f(x) = \frac{1}{x-1}$ on the domain $[-2, 6]$.

1. Although $f(x)$ has a discontinuity at $x = 1$, a single call to `plt.plot()` in the usual way will make the curve look continuous. Split up the domain into $[-2, 1)$ and $(1, 6]$. Plot the two sides of the curve separately so that the graph looks discontinuous at $x = 1$.
2. Plot both curves with a dashed magenta line. Set the keyword argument `linewidth` (or `lw`) of `plt.plot()` to 4 to make the line a little thicker than the default setting.
3. Use `plt.xlim()` and `plt.ylim()` to change the range of the x-axis to $[-2, 6]$ and the range of the y-axis to $[-6, 6]$.

The plot should resemble the figure below.



Figures, Axes, and Subplots

The window that `plt.show()` reveals is called a *figure*, stored in Python as a `plt.Figure` object. A space on a figure where a plot is drawn is called an *axes*, a `plt.Axes` object. A figure can have multiple axes, and a single program may create several figures. There are several ways to create or grab figures and axes with `plt` functions.

Function	Description
`axes()`	Add an axes to the current figure
`figure()`	Create a new figure or grab an existing figure
`gca()`	Get the current axes
`gcf()`	Get the current figure
`subplot()`	Add a single subplot to the current figure
`subplots()`	Create a figure and add several subplots to it

Usually when a figure has multiple axes, they are organized into non-overlapping *subplots*. The command `plt.subplot(nrows, ncols, plot_number)` creates an axes in a subplot grid where `nrows` is the number of rows of subplots in the figure, `ncols` is the number of columns, and `plot_number` specifies which subplot to modify. If the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.

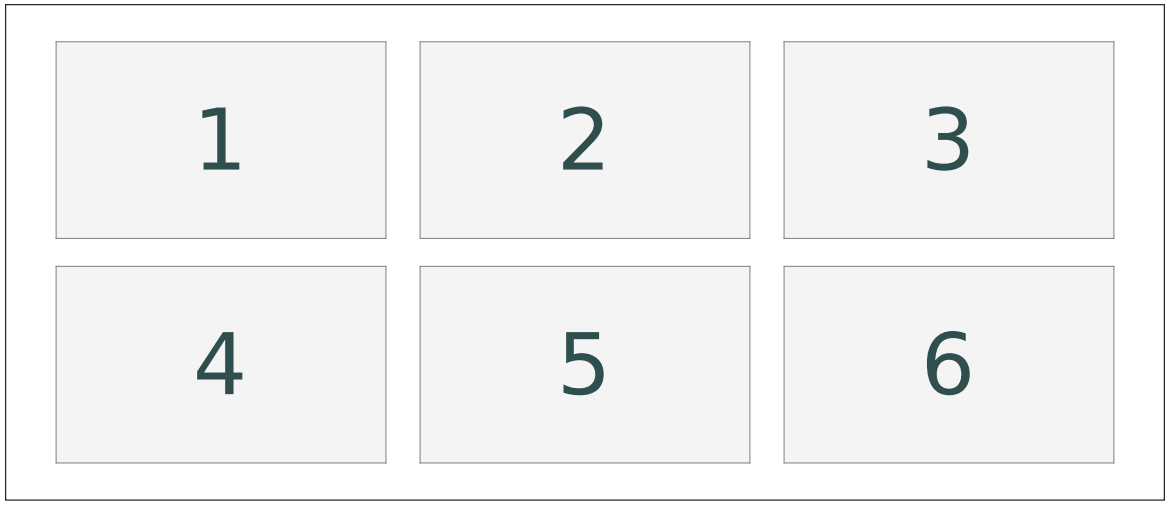
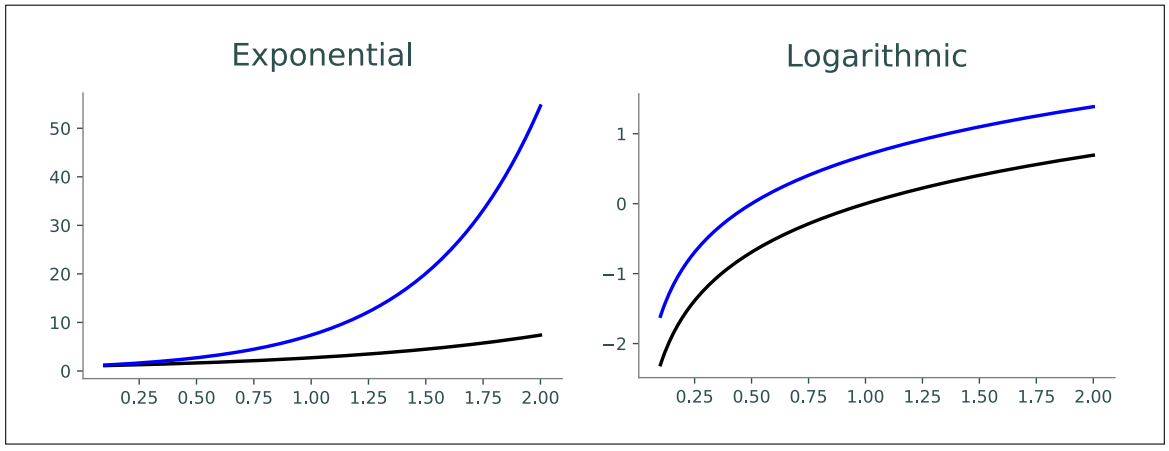


Figure 5.3: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

```
>>> x = np.linspace(.1, 2, 200)
# Create a subplot to cover the left half of the figure.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, np.exp(x), 'k', lw=2)
>>> ax1.plot(x, np.exp(2*x), 'b', lw=2)
>>> plt.title("Exponential", fontsize=18)

# Create another subplot to cover the right half of the figure.
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, np.log(x), 'k', lw=2)
>>> ax2.plot(x, np.log(2*x), 'b', lw=2)
>>> ax2.set_title("Logarithmic", fontsize=18)
>>> plt.show()
```



Note

Plotting functions such as `plt.plot()` are shortcuts for accessing the current axes on the current figure and calling a method on that `Axes` object. Calling `plt.subplot()` changes the current axis, and calling `plt.figure()` changes the current figure. Use `plt.gca()` to get the current axes and `plt.gcf()` to get the current figure. Compare the following equivalent strategies for producing a figure with two subplots.

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

Problem 4. Write a function that plots the functions $\sin(x)$, $\sin(2x)$, $2\sin(x)$, and $2\sin(2x)$ on the domain $[0, 2\pi]$, each in a separate subplot of a single figure.

1. Arrange the plots in a 2×2 grid of subplots.
2. Set the limits of each subplot to $[0, 2\pi] \times [-2, 2]$.
   (Hint: Consider using `plt.axis([xmin, xmax, ymin, ymax])` instead of `plt.xlim()` and `plt.ylim()` to set all boundaries simultaneously.)
3. Use `plt.title()` or `ax.set_title()` to give each subplot an appropriate title.
4. Use `plt.suptitle()` or `fig.suptitle()` to give the overall figure a title.
5. Use the following colors and line styles.

   $\sin(x)$: green solid line. $\sin(2x)$: red dashed line.

   $2\sin(x)$: blue dashed line. $2\sin(2x)$: magenta dotted line.

ACHTUNG!

Be careful not to mix up the following functions.

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` (or `ax.axis()`) sets properties of the x- and y-axis in the current axes, such as the x and y limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

Other Kinds of Plots

Line plots are not always the most illuminating choice of graph to describe a set of data. Matplotlib provides several other easy ways to visualize data.

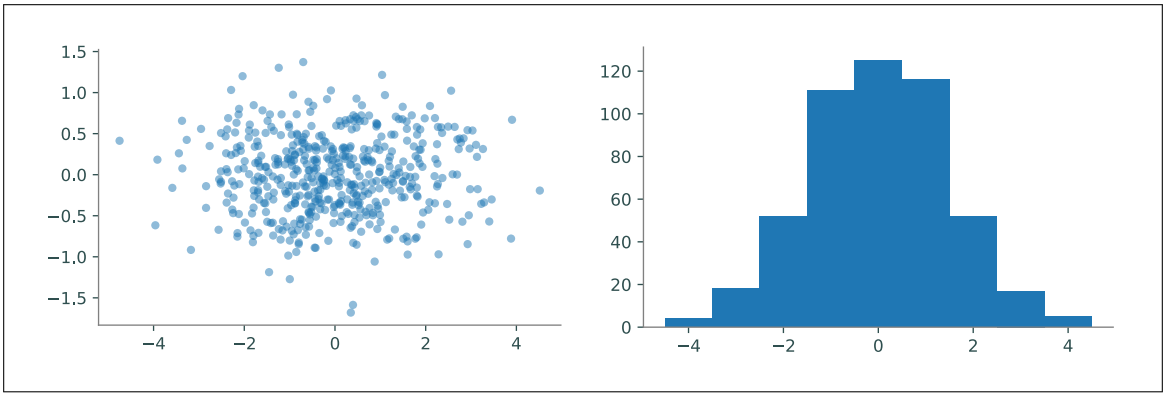
- A *scatter plot* plots two 1-dimensional arrays against each other without drawing lines between the points. Scatter plots are particularly useful for data that is not correlated or ordered.

  To create a scatter plot, use `plt.plot()` and specify a point marker (such as `'o'` or `'*'`) for the line style, or use `plt.scatter()` (or `ax.scatter()`). Beware that `plt.scatter()` has slightly different arguments and syntax than `plt.plot()`.

- A *histogram* groups entries of a 1-dimensional data set into a given number of intervals, called *bins*. Each bin has a bar whose height indicates the number of values that fall in the range of the bin. Histograms are best for displaying distributions, relating data values to frequency.

  To create a histogram, use `plt.hist()` (or `ax.hist()`). Use the argument `bins` to specify the edges of the bins or to choose a number of bins. The `range` argument specifies the outer limits of the first and last bins.

```
# Get 500 random samples from two normal distributions.
>>> x = np.random.normal(scale=1.5, size=500)
>>> y = np.random.normal(scale=0.5, size=500)

# Draw a scatter plot of x against y, using transparent circle markers.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, y, 'o', markersize=5, alpha=.5)

# Draw a histogram to display the distribution of the data in x.
>>> ax2 = plt.subplot(122)
>>> ax2.hist(x, bins=np.arange(-4.5, 5.5))      # Or, equivalently,
#   ax2.hist(x, bins=9, range=[-4.5, 4.5])

>>> plt.show()
```



Problem 5. The Fatality Analysis Reporting System (FARS) is a nationwide census that provides yearly data regarding fatal injuries suffered in motor vehicle traffic crashes.[a] The array contained in `FARS.npy` is a small subset of the FARS database from 2010–2014. Each of the 148,206 rows in the array represents a different car crash; the columns represent the hour (in military time, as an integer), the longitude, and the latitude, in that order.

Write a function to visualize the data in `FARS.npy`. Use `np.load()` to load the data, then create a single figure with two subplots:

1. A scatter plot of longitudes against latitudes. Because of the large number of data points, use black pixel markers (use `"k,"` as the third argument to `plt.plot()`). Label both axes using `plt.xlabel()` and `plt.ylabel()` (or `ax.set_xlabel()` and `ax.set_ylabel()`). (Hint: Use `plt.axis("equal")` or `ax.set_aspect("equal")` so that the x- and y-axis are scaled the same way.

2. A histogram of the hours of the day, with one bin per hour. Set the limits of the x-axis appropriately. Label the x-axis. You should be able to clearly see which hours of the day experience more traffic.

[a] See `http://www.nhtsa.gov/FARS`.

Matplotlib also has tools for creating other kinds of plots for visualizing 1-dimensional data, including bar plots and box plots. See the Matplotlib Appendix for examples and syntax.

Visualizing 3-D Surfaces

Line plots, histograms, and scatter plots are good for visualizing 1- and 2-dimensional data, including the domain and range of a function $f : \mathbb{R} \to \mathbb{R}$. However, visualizing 3-dimensional data or a function $g : \mathbb{R}^2 \to \mathbb{R}$ (two inputs, one output) requires a different kind of plot. The process is similar to creating a line plot but requires slightly more setup: first construct an appropriate domain, then calculate the image of the function on that domain.

NumPy's `np.meshgrid()` function is the standard tool for creating a 2-dimensional domain in the Cartesian plane. Given two 1-dimensional coordinate arrays, `np.meshgrid()` creates two corresponding coordinate matrices. See Figure 5.6.

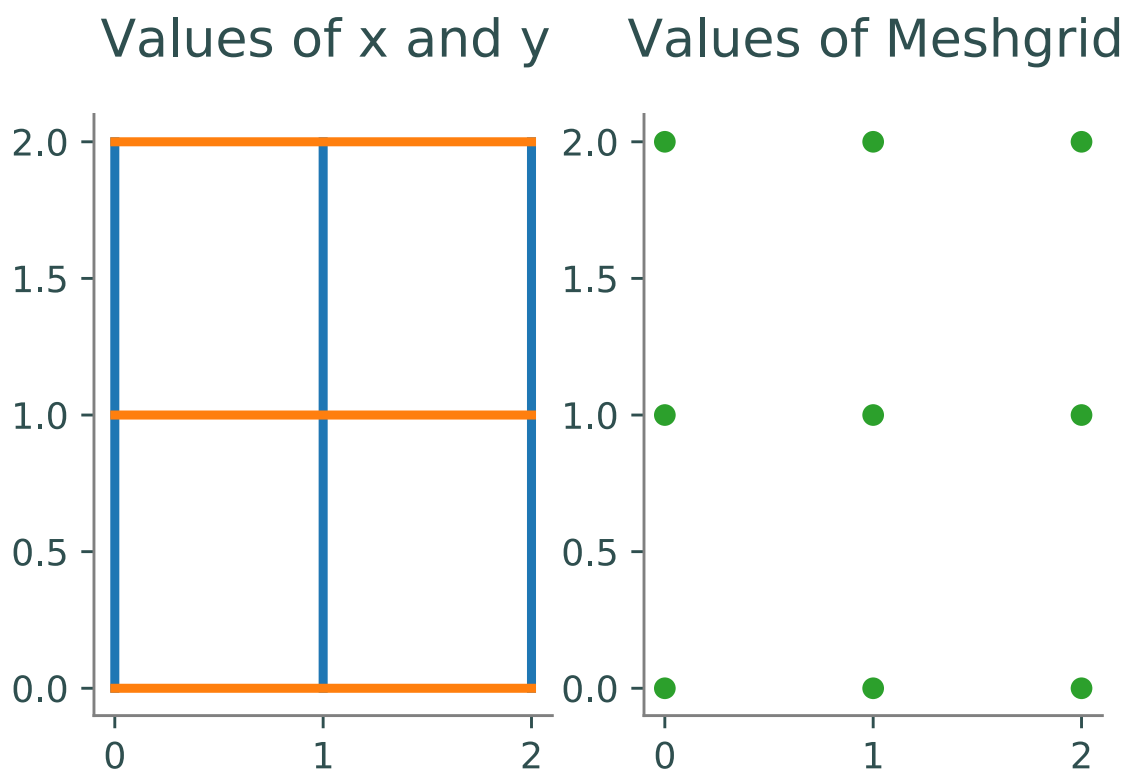


Figure 5.6: In the left plot, we have two arrays where x and y have the values `x = y = [0,1,2]`. The command `np.meshgrid(x, y)` returns the arrays $X = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$ and $Y = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$. These give the x- and y-coordinates of the points in the grid formed by `x` and `y` as seen in the right plot and satisfy `(X[i,j], Y[i,j]) = (x[i],y[j])`.

```
>>> x, y = [0, 1, 2], [3, 4, 5]      # A rough domain over [0,2]x[3,5].
>>> X, Y = np.meshgrid(x, y)         # Combine the 1-D data into 2-D data.
>>> for xrow, yrow in zip(X,Y):
...     print(xrow, yrow, sep='\t')
...
[0 1 2]    [3 3 3]
[0 1 2]    [4 4 4]
[0 1 2]    [5 5 5]
```

With a 2-dimensional domain, $g(x, y)$ is usually visualized with two kinds of plots.

- A *heat map* assigns a color to each point in the domain, producing a 2-dimensional colored picture describing a 3-dimensional shape. Darker colors typically correspond to lower values while lighter colors typically correspond to higher values.

Use `plt.pcolormesh()` to create a heat map. It is required to include an argument for the shading type; this determines the layout and fill style of the heat map. Setting `shading='auto'` will automatically choose a fill method suited to the data being graphed.

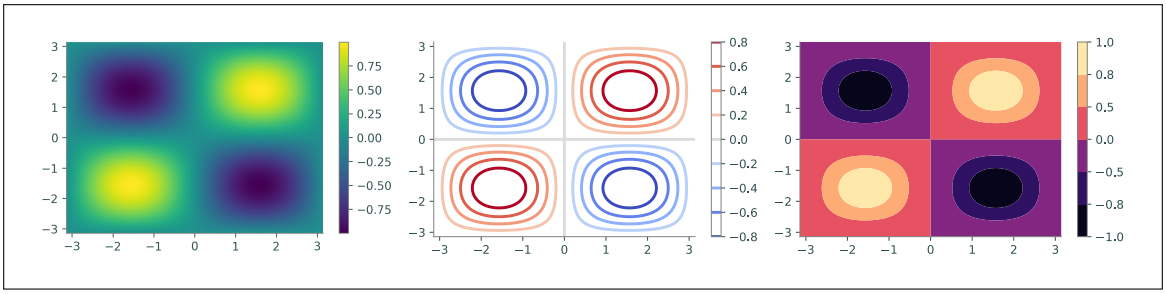
- A *contour map* draws several *level curves* of g on the 2-dimensional domain. A level curve corresponding to the constant c is the collection of points $\{(x, y) \mid c = g(x, y)\}$. Coloring the space between the level curves produces a discretized version of a heat map. Including more and more level curves makes a filled contour plot look more and more like the complete, blended heat map.

  Use `plt.contour()` to create a contour plot and `plt.contourf()` to create a filled contour plot. Specify either the number of level curves to draw, or a list of constants corresponding to specific level curves.

These functions each receive the keyword argument `cmap` to specify a color scheme (some of the better schemes are `"viridis"`, `"magma"`, and `"coolwarm"`). For the list of all Matplotlib color schemes, see `http://matplotlib.org/examples/color/colormaps_reference.html`.

Finally, `plt.colorbar()` draws the color scale beside the plot to indicate how the colors relate to the values of the function.

```
# Create a 2-D domain with np.meshgrid().
>>> x = np.linspace(-np.pi, np.pi, 100)
>>> y = x.copy()
>>> X, Y = np.meshgrid(x, y)
>>> Z = np.sin(X) * np.sin(Y)       # Calculate g(x,y) = sin(x)sin(y).

# Plot the heat map of f over the 2-D domain.
>>> plt.subplot(131)
>>> plt.pcolormesh(X, Y, Z, cmap="viridis", shading="auto")
>>> plt.colorbar()
>>> plt.xlim(-np.pi, np.pi)
>>> plt.ylim(-np.pi, np.pi)

# Plot a contour map of f with 10 level curves.
>>> plt.subplot(132)
>>> plt.contour(X, Y, Z, 10, cmap="coolwarm")
>>> plt.colorbar()

# Plot a filled contour map, specifying the level curves.
>>> plt.subplot(133)
>>> plt.contourf(X, Y, Z, [-1, -.8, -.5, 0, .5, .8, 1], cmap="magma")
>>> plt.colorbar()
>>> plt.show()
```



Problem 6. Write a function to plot $g(x, y) = \frac{\sin(x)\sin(y)}{xy}$ on the domain $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$.

1. Create 2 subplots: one with a heat map of g, and one with a contour map of g. Choose an appropriate number of level curves, or specify the curves yourself.
2. Set the limits of each subplot to $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$.
3. Choose a non-default color scheme.
4. Include a color scale bar for each subplot.

Additional Material

Further Reading and Tutorials

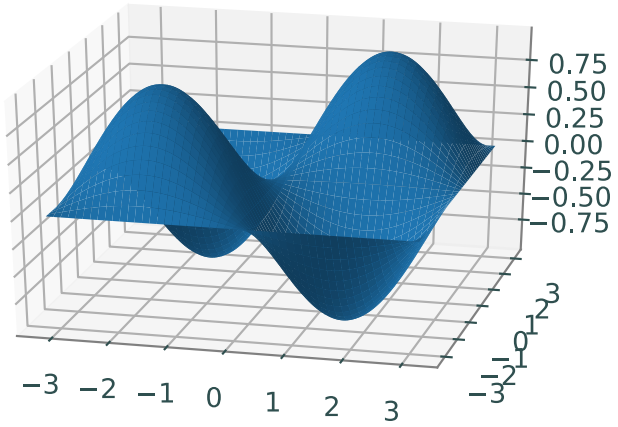
Plotting takes some getting used to. See the following materials for more examples.

- https://www.labri.fr/perso/nrougier/teaching/matplotlib/.
- https://matplotlib.org/users/pyplot_tutorial.html.
- http://scipy-lectures.org/intro/matplotlib/.
- The Matplotlib Appendix in this manual.

3-D Plotting

Matplotlib can also be used to plot 3-dimensional surfaces. The following code produces the surface corresponding to $g(x, y) = \sin(x)\sin(y)$.

```
# Create the domain and calculate the range like usual.
>>> x = np.linspace(-np.pi, np.pi, 200)
>>> y = np.copy(x)
>>> X, Y = np.meshgrid(x, y)
>>> Z = np.sin(X) * np.sin(Y)

# Draw the corresponding 3-D plot using some extra tools.
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1,1,1, projection='3d')
>>> ax.plot_surface(X, Y, Z)
>>> plt.show()
```



Animations

Lines and other graphs can be altered dynamically to produce animations. Follow these steps to create a Matplotlib animation:

1. Calculate all data that is needed for the animation.
2. Define a figure explicitly with `plt.figure()` and set its window boundaries.
3. Draw empty objects that can be altered dynamically.
4. Define a function to update the drawing objects.
5. Use `matplotlib.animation.FuncAnimation()`.

The submodule `matplotlib.animation` contains the tools for putting together and managing animations. The function `matplotlib.animation.FuncAnimation()` accepts the figure to animate, the function that updates the figure, the number of frames to show before repeating, and how fast to run the animation (lower numbers mean faster animations).

```
from matplotlib.animation import FuncAnimation

def sine_animation():
    # Calculate the data to be animated.
    x = np.linspace(0, 2*np.pi, 200)[:-1]
    y = np.sin(x)

    # Create a figure and set the window boundaries of the axes.
    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
    plt.ylim(-1.2, 1.2)

    # Draw an empty line. The comma after 'drawing' is crucial.
    drawing, = plt.plot([],[])

    # Define a function that updates the line data.
    def update(index):
        drawing.set_data(x[:index], y[:index])
        return drawing,                    # Note the comma!

    a = FuncAnimation(fig, update, frames=len(x), interval=10)
    plt.show()
```

Try using the following function in place of `update()`. Can you explain why this animation is different from the original?

```
def wave(index):
    drawing.set_data(x, np.roll(y, index))
    return drawing,
```

To animate multiple objects at once, define the objects separately and make sure the update function returns both objects.

```
def sine_cosine_animation():
    x = np.linspace(0, 2*np.pi, 200)[:-1]
    y1, y2 = np.sin(x), np.cos(x)

    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
    plt.ylim(-1.2, 1.2)

    sin_drawing, = plt.plot([],[])
    cos_drawing, = plt.plot([],[])

    def update(index):
        sin_drawing.set_data(x[:index], y1[:index])
        cos_drawing.set_data(x[:index], y2[:index])
        return sin_drawing, cos_drawing,

    a = FuncAnimation(fig, update, frames=len(x), interval=10)
    plt.show()
```

Animations can also be 3-dimensional. The only major difference is an extra operation to set the 3-dimensional component of the drawn object. The code below animates the space curve parametrized by the following equations:

$$x(\theta) = \cos(\theta)\cos(6\theta), \qquad y(\theta) = \sin(\theta)\cos(6\theta), \qquad z(\theta) = \tfrac{\theta}{10}$$

```
def rose_animation_3D():
    theta = np.linspace(0, 2*np.pi, 200)
    x = np.cos(theta) * np.cos(6*theta)
    y = np.sin(theta) * np.cos(6*theta)
    z = theta / 10

    fig = plt.figure()
    ax = fig.gca(projection='3d')           # Make the figure 3-D.
    ax.set_xlim3d(-1.2, 1.2)                # Use ax instead of plt.
    ax.set_ylim3d(-1.2, 1.2)
    ax.set_aspect("equal")

    drawing, = ax.plot([],[],[])            # Provide 3 empty lists.

    # Update the first 2 dimensions like usual, then update the 3-D component.
    def update(index):
        drawing.set_data(x[:index], y[:index])
        drawing.set_3d_properties(z[:index])
        return drawing,

    a = FuncAnimation(fig, update, frames=len(x), interval=10, repeat=False)
    plt.show()
```