Appendix F
Matplotlib

John D. Hunter, a neurobiologist, began developing matplotlib (`https://matplotlib.org/`) in 2003, originally to emulate Mathworks' MATLAB. Since then it has grown into an amazing plotting library, backed by a large community of developers.

One issue when learning matplotlib is that there are two main ways to write code (and matplotlib's documentation and examples use both):

(1) Explicitly initialize variables representing the chart figure and axes, and call functions on them:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
xs = [2, 4, 6, 8]
ys = [3, 5, 7, 9]
ax.plot(xs, ys)
ax.set_title('Simple plot')
plt.show()
```

Listing F.1. Plot a line using an explicit figure and axis

There are several ways to create these variables:

```
fig = plt.figure()
ax = fig.subplots()
```

or

```
ax = plt.gca()   # gc means "get current"
fig = plt.gcf()
```

(2) Create a chart using the functions in the `pyplot` submodule (`https://matplotlib.org/stable/tutorials/pyplot.html`):

```
import matplotlib.pyplot as plt

xs = [2, 4, 6, 8]
ys = [3, 5, 7, 9]
plt.plot(xs, ys)
plt.title('Simple plot')
plt.show()
```

Listing F.2. Plot a line using pyplot

We've used this second style in this text, except in a few cases.

Needless to say, Listing F.1 (simpleFigAx.py) and Listing F.2 (simplePyPlot.py) both produce the same chart (Fig. F.1).

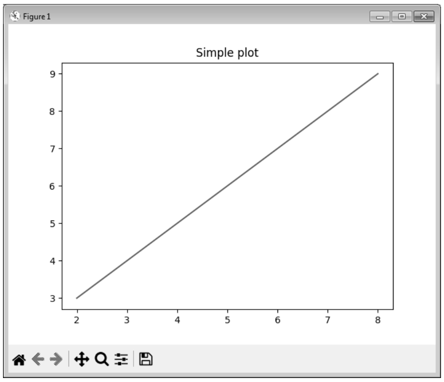

Figure F.1. A line chart

It's worth noting the buttons at the bottom of every matplotlib window. They offer useful features like panning, zooming, and saving of the graph.

There's an older coding style (now deprecated) based around the `pylab` interface, which can still be found in some textbooks and online:

```
from pylab import *
xs = [2, 4, 6, 8]
ys = [3, 5, 7, 9]
plot(xs, ys)
ptitle('Simple plot')
show()
```

Programs of this type can usually be easily edited to use `pyplot` functions instead.

Matplotlib is built on top of numpy, and so a great number of examples (including many in matplotlib's own documentation) involve the plotting of data stored in numpy arrays:

```
import numpy as np
x = np.array([1, 2, 3, 4])
y = x*2
plt.plot(x,y)
```

We prefer to stick with Python lists where possible, although numpy becomes inescapable when dealing with 3D graphs, as we'll see later.

The matplotlib online documentation is excellent. Good starting points include:

- The quick start guide: `https://matplotlib.org/stable/users/explain/quick_start.html`
- The user guide: `https://matplotlib.org/stable/users/`
- The tutorials page: `https://matplotlib.org/stable/tutorials/`
- A collection of printable 'cheatsheets': `https://matplotlib.org/cheatsheets/`
- A page listing the main plot types (`https://matplotlib.org/stable/plot_types/`) in five categories: pairwise data (i.e. graphing (x,y) coordinates), statistical distributions, gridded data, irregularly gridded data, and 3D and volumetric data.
- An official gallery of examples: `https://matplotlib.org/stable/gallery/`
- An unofficial gallery of examples: `https://python-graph-gallery.com/best-python-chart-examples/`

F.1 The Big Three

The 'big three' are line plots, scatter plots, and bar charts.

F.1.1 The Line Plot. Listing F.2 (simplePyPlot.py) above shows how to draw a line using lists of coordinates. More commonly these are generated using an equation, as in Listing F.3 (parabola.py):

```
import matplotlib.pyplot as plt
from frange import *

xs = linspace(-2, 2, 100)
ys = [x**2 for x in xs]
plt.plot(xs, ys, label="y=x^2")
plt.title("Parabola")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

Listing F.3. Plot a parabola

We've used our linspace() function imported from frange.py, but you can utilize numpy's linspace() instead if you wish. It generates the requested number of floats equally spaced out across a specified range. For example:

```
>>> from frange import *
>>> linspace(-2, 2, 10)
[-2.0, -1.5555555555555556, -1.1111111111111112,
-0.6666666666666667, -0.2222222222222232,
0.2222222222222232, 0.6666666666666665,
1.1111111111111107, 1.5555555555555554, 2.0]
```

The plot (Fig. F.2) includes a label, drawn on the graph using plt.legend(), and the x- and y- axes are labeled.

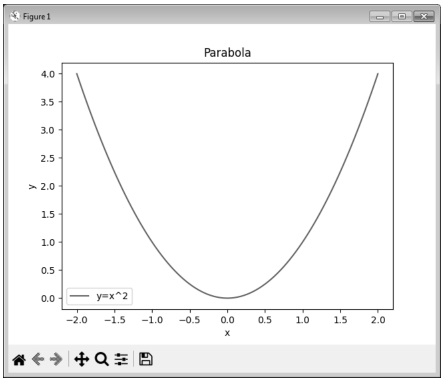

Figure F.2. A line plot of a parabola

F.1.2 The Scatter Plot. Scatter plots also utilize lists of coordinates, but plotted with scatter().

```
import random
import matplotlib.pyplot as plt

xs = [ random.gauss(0, 1) for _ in range(1000)]
ys = [ random.gauss(0, 1) for _ in range(1000)]
plt.scatter(xs, ys, s=5)  # s is point size
plt.show()
```

Listing F.4. A Gaussian scattering

The graph produced by Listing F.4 (randScatter.py) is shown in Fig. F.3 (from now on we'll crop the matplotlib buttons to reduce the image size).

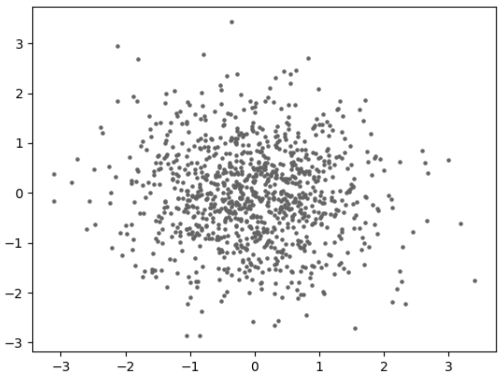

Figure F.3. A scatter plot of Gaussian numbers

F.1.3 The Bar Chart. Listing F.5 (days13.py) uses a bar chart to shows the day distribution of the 13th across the months in the years 1800 to 2201. bar() is supplied with the bar names and a list of heights.

```
from datetime import date
import matplotlib.pyplot as plt


WEEKDAYS = ['Mon','Tue','Wed','Thur','Fri','Sat','Sun']
days13 = [0]*7
for year in range(1800, 2201):
  for month in range(1,13):
    days13[ date(year, month, 13).weekday()] += 1
print(*days13)
plt.bar(WEEKDAYS, days13)  # labels, heights
plt.title('The 13th Days')
plt.show()
```

Listing F.5. A tally of days numbered 13

Fig. F.4 is far from ideal, and we'll look at how to improve it in a later section.

F.2 Common Chart Variants

There are three common chart variants: plotting several curves in a single chart (to aid their comparison), adding error bars to line and bar plots, and changing one or both axes to use a logarithmic scale.

F.2.1 Multiple Curves in a Single Graph. Listing F.6 (taylorcos.py) draws the cosine curve and two Taylor series approximations of degrees 2 and 4 in the same chart (Fig. F.5).

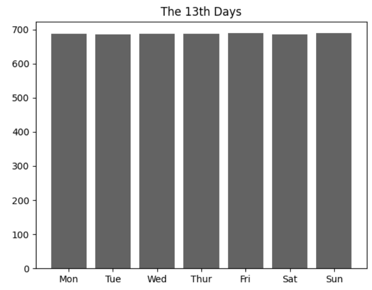

Figure F.4. A bar chart of 13th's

```
xs = linspace(-6, 6, 50)

# Plot y = cos(x)
ys = [ math.cos(x) for x in xs]
plt.plot(xs, ys, 'b', label ='cos(x)')  # solid blue

# Plot degree 2 Taylor polynomial
ys2 = [ (1 - x**2/2) for x in xs]
plt.plot(xs, ys2, 'r-.', label ='Degree 2')  # red dot dashes

# Plot degree 4 Taylor polynomial
ys4 = [(1 - x**2/2 + x**4/24) for x in xs]
plt.plot(xs, ys4, 'g:', label ='Degree 4')  # green dots

plt.grid(True, linestyle =':')  # a dotted grid pattern
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.xlim([-6, 6])
plt.ylim([-4, 4])

plt.legend()
plt.title('Taylor Polynomials of cos(x) at x = 0')
plt.show()
```

Listing F.6. Cosine and approximations

There are three calls to plot(), and to distinguish between the drawn curves we've used different line colors, patterns, and labels, which are printed on the graph by legend(). Two other additions are the use of grid lines and the limiting of the x- and y- axis ranges.

F.2.2 Error Bars. Error bars are most often added to line plots and bar charts. Since the coding is similar in both cases, we'll just explain the parabolic line graph here (Listing F.7; errParabola.py).

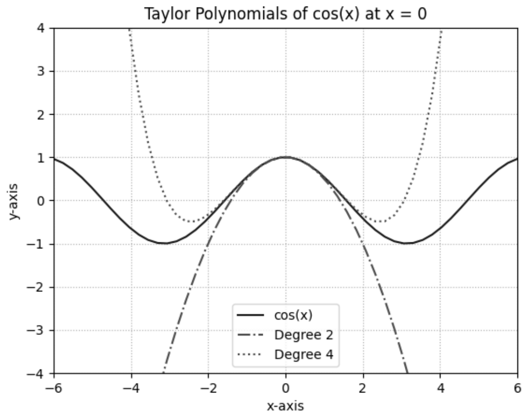

Figure F.5. Multiple lines in a graph

```
n = 10
xs = linspace(-2, 2, n)
ys = [x**2 for x in xs]
plt.plot(xs, ys, label="y=x^2")

yErrs = linspace(0.2, 0.5, n)
plt.errorbar(xs, ys, yerr = yErrs, fmt ='o')

plt.title("Parabola with Error Bars")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

Listing F.7. A parabola with error bars

The lengths of the error bars are passed to errorbar() along with how the error symbol should be drawn (see Fig. F.6).

F.2.3 Logarithmic Axes. Listing F.8 (loglog.py) contains three alternative ways to plot the same coordinates – using plot() as seen previously, and also calls to semilogx(), semilogy(), and loglog() which automatically scale the data on the x-axis, or y-axis, or both. Note that there's no need to modify the coordinate data.

```
xs = linspace(1, 10, 500)
ys = [x**3 for x in xs]
# plt.plot(xs, ys)

# plt.semilogx(xs, ys)  # x-axis log
# plt.semilogy(xs, ys)  # y-axis log
plt.loglog(xs, ys)   # both axes use logs
```

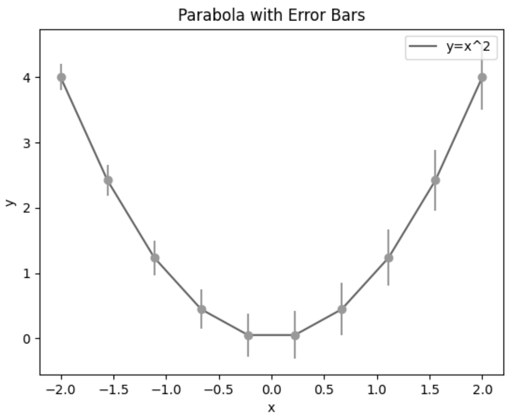

Figure F.6. Error bars on a parabola

```
plt.title("Log plot")
plt.show()
```

Listing F.8. Log scaling of axes

F.3 Customizing a Plot

There are many, many ways to adjust how graph elements are drawn. When looking for a particular matplotlib function for the task, it helps to know what an element is called. Fig. F.7, taken from the matplotlib documentation, lists the main ones.

Listing F.9 (parabolaCustom.py) includes the adjustments that we use most frequently in the main text.

```
xs = linspace(-2, 2, 10)
ys = [x**2 for x in xs]

fig = plt.figure(figsize = (10, 8)) # bigger window

plt.axis('equal')  # same scaling used on both axes
plt.xlim(-2.5, 2.5)
plt.minorticks_on()
plt.xticks( list(frange(-2.5, 3, 0.5)))

plt.axhline(0, color="Green")  # horizontal line
plt.axvline(0, color="Green")  # vertical line

plt.plot(xs, ys, alpha = 0.4, label ='$y = x^2$', # Latex
```

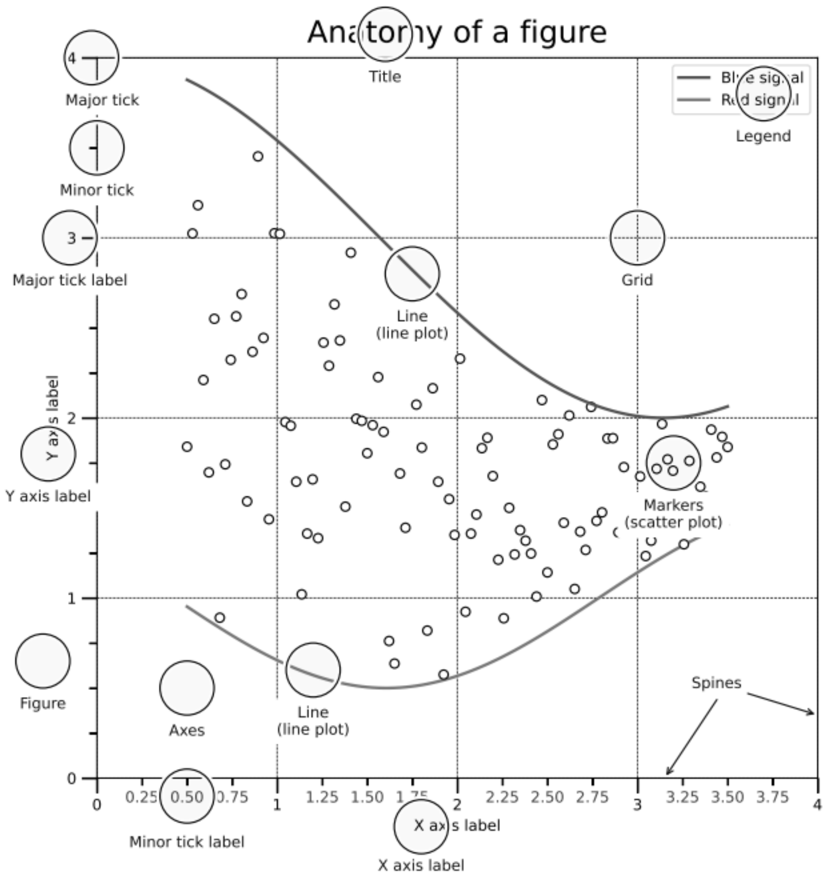

Figure F.7. Names of figure elements

```
                color ='red', linestyle ='--',
                linewidth = 2, marker ='D',
                markersize = 5, mfc ='blue', mec ='blue')
                      # marker face and edge colors

plt.title('Parabola plot', fontsize = 20) # bigger title
plt.xlabel('x')
plt.ylabel('y')
fig.canvas.manager.set_window_title('Parabola plot')
             # give the window title bar a better name

fig.text(0.8, 0.15, 'Some info',
     fontsize = 12, color ='green',
     ha ='right', va ='bottom',
     alpha = 0.7)    # draw text at (0.8, 0.15) offset

plt.grid(alpha =.6, linestyle ='--')  # grid lines
plt.legend(loc="upper left")      # move the legend

print("Saving plot to savedplot.png")
plt.savefig("savedplot.png", dpi=300)

plt.show()
```

Listing F.9. Customization code

It helps to cross-reference the source with what it generates in Fig. F.8.

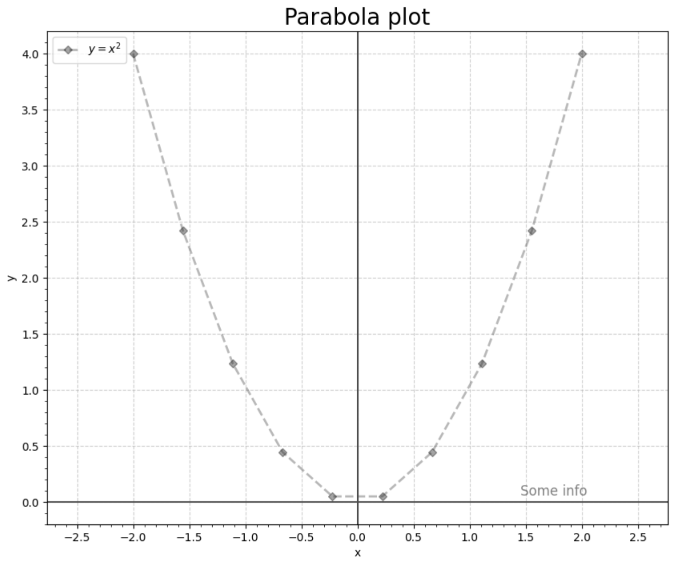

Figure F.8. Customization results

Perhaps the hardest customizations to remember are the marker line and point styles (e.g. '–' and 'D' in Listing F.9). The common ones are listed in Tables F.1 and F.2.

The bar chart back in Fig. F.4 is hard to read, but there are two customizations that can improve its look: the y-axis doesn't need to start at 0, and the bars can be labeled with their numerical values. The relevant lines are shown in Listing F.10 (daysCustom.py), and the resulting graph is Fig. F.9.

```
MIN = 680
bots=[MIN]*7
```

Label	Style
-	Solid
–	Dashed
:	Dotted
-.	Dash-dot

Table F.1. Marker line styles

Label	Marker	Style
.	·	Point
o	∘	Circle
+	+	Plus
x	×	Cross
D	⋄	Diamond
v	▽	Downward Triangle
∧	△	Upward Triangle
s	□	Square
*	★	Star

Table F.2. Marker point styles

```
ds = [val-MIN for val in days13]   # cut off the heights

plt.bar(WEEKDAYS, ds, width=0.4, bottom=bots)
                           # set vertical baseline

xlocs, xlabs = plt.xticks()
for i, v in enumerate(days13):
  plt.text(xlocs[i] - 0.2, v + 0.08, str(v))
```

Listing F.10. Customizing a bar chart

The trickiest part of the code is determining the positions of the labels that appear above the bars. The x and y values are obtained in different ways. The x is retrieved via a call to xticks() which returns the positions of the tick marks on the x-axis. The y value is calculated from the heights in the `days13` list. Both are slightly modified so that the text is better positioned.

Another complication is that the heights of the bars on the graph do *not* come from `days13`, but from the modified `ds` list which holds smaller heights. The bars are drawn correctly since the bar() call includes a bottom argument which sets the start of the y-axis.

A minor change is that we prefer the bars to be narrower, which is achieved by setting the width argument in bar().

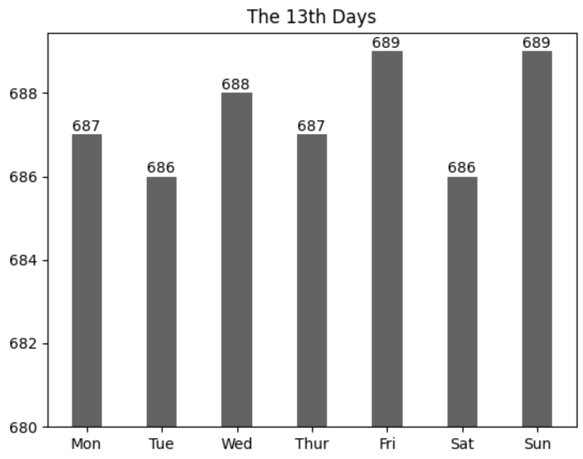

Figure F.9. Customized bar chart of 13th's

F.4 Other Useful Plot Types

Matplotlib supports a very large variety of plot types (e.g look at `https://matplotlib.org/stable/plot_types/`). In this section, we'll give brief examples of five that we've used – the histogram, the box plot, the polar chart, the contour lines plot, and the heat map.

F.4.1 The Histogram. Listing F.11 (heights.py) reads in the heights of high school boys and girls from a CSV file, and creates two box plots. The means and standard deviations of the two groups are also reported as vertical lines – a solid line for the mean, and two dashed lines for mean ± stdev. The graph is shown in Fig. F.10.

Listing F.11 utilizes buildDist() to load data from the CSV file into a dictionary of key-list pairs. The keys are the data's column names, and each list holds all the values from a column as floats.

```
def buildDist(fnm):
  columns = defaultdict(list)
  with open(fnm) as f:
    reader = csv.DictReader(f)
    for row in reader:
      # read a row as {column1: value1, column2: value2,...}
      for (k,v) in row.items():
        columns[k].append(float(v))
  return columns

def stats(label, heights, col):
  # report the mean and stdev for the supplied heights data
```

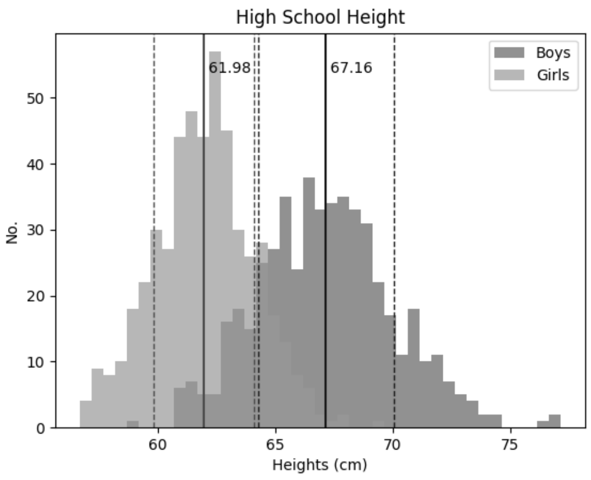

Figure F.10. Histograms of boys and girls heights

```
  mean = statistics.mean(heights)
  stdev = statistics.stdev(heights)
  print("Statistics ("+ str(label) +")")
  print(f"Mean: {mean:.2f} ; stdev: {stdev:.2f}")
  minY, maxY = plt.ylim()
  plt.axvline(mean, color=col)
  plt.text(mean, maxY*0.9, f" {mean:.2f}")
  plt.axvline(mean+stdev, color=col, ls='dashed', lw=1)
  plt.axvline(mean-stdev, color=col, ls='dashed', lw=1)

# load data
columns =  buildDist('hs_heights_pair.csv')
boys = columns['boys']  # use CSV data column labels
girls = columns['girls']

# calculate a range for the box plots bins, and
# split into boxes of width 0.5
minHeight = min([min(boys),min(girls)])
maxHeight = max([max(boys),max(girls)])
bins = list(frange(minHeight, maxHeight+0.5, 0.5))

plt.hist(boys, bins, label="Boys", alpha=0.7)
plt.hist(girls, bins, label="Girls", alpha=0.7)

# report the means and stdevs for the two groups
stats("Boys", boys, "blue")
stats("Girls", girls, "green")

plt.xlabel('Heights (cm)')
plt.ylabel('No.')
plt.title('High School Height')
plt.legend()
plt.show()
```

Listing F.11. Building a histogram

The CSV file, `hs_heights_pair.csv`, begins like so:

```
boys,girls
64.44,62.47
67.4,63.27
63.93,62.99
   : # many more lines, not shown
```

This format means that the columns dictionary returned by buildDist() will contains two lists, `boys` and `girls`.

A difficult aspect of histograms can be how to decide on the bins range, and the size of each bin. Listing F.11 automates the first task by finding the minimum and maximum values in the two data lists; however, the bin size is fixed at 0.5.

It's often useful to include mean and standard derivation information with a histogram, which is handled by stats(). It prints the information to standard output, and draws lines on the graph along with the mean values.

F.4.2 The Box Plot. A box plot summarizes data having a minimum, first quartile, median, third quartile, and maximum. These elements are illustrated in Fig. F.11.

A box extends from the first data quartile (Q1) to the third (Q3), which is termed the Interquartile Range (IQR). A line goes through the box to mark the median, and 'whisker' lines extend from the top and bottom to 'maximum' and 'minimum' values. These aren't the smallest and largest numbers in the data, but positions marking Q1 - 1.5*IQR and Q3 + 1.5*IQR, with any data outside these ranges being classed as outliers. The display of the mean is an optional extra, but sometimes useful.

The output from Listing F.12 (boxplots.py) is shown in Fig. F.12.

```
data1 = [ random.gauss(100, 10) for _ in range(200)]
data2 = [ random.gauss(90, 20) for _ in range(200)]
data3 = [ random.gauss(80, 30) for _ in range(200)]
data4 = [ random.gauss(70, 40) for _ in range(200)]
ds = [data1, data2, data3, data4]

plt.boxplot(ds, showmeans=True)  # include means
plt.xticks([1, 2, 3, 4], ["Jim", "John", "Joan", "Jane"])
                          # change the x-axis labels
plt.show()
```

Listing F.12. Drawing box plots

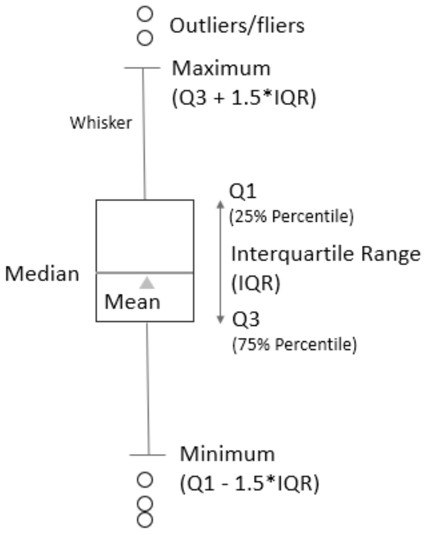

Figure F.11. Elements of a box plot

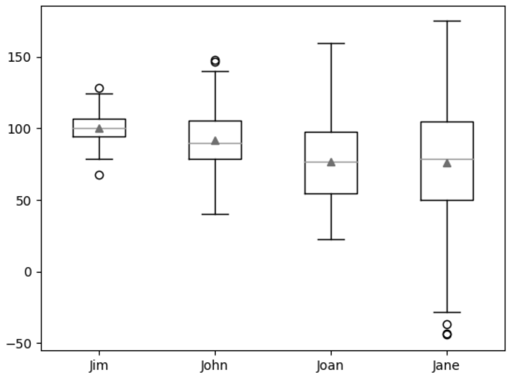

Figure F.12. Student box plots

F.4.3 The Polar Chart. Polar plots utilize the polar() function which is passed lists of θs and radii. Listing F.13 (butterfly.py) generates the butterfly curve shown in Fig. F.13.

```
n = 5 # no. of loops
# generate radii from angles
thetas = linspace(0, 2*math.pi*n, 100*n)
```

```
rs = [ math.exp(math.sin(theta)) - 2*math.cos(4*theta) +
       math.sin(theta/12)**5
                     for theta in thetas]
plt.polar(thetas, rs)
plt.title("Butterfly Curve")
plt.show()
```

Listing F.13. Generating a butterfly

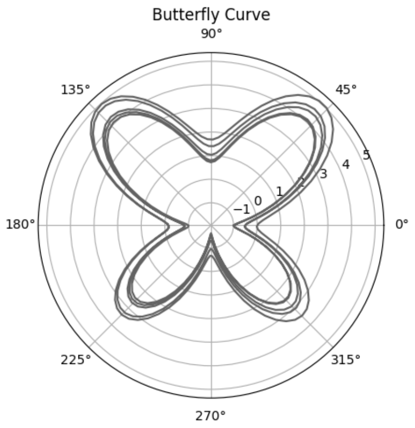

Figure F.13. The butterfly curve

Listing F.14 (spiral.py) shows how to customize the polar axes (e.g. reduce the number of radial ticks and move the radial labels). This requires access to the axis variable, which means that the polar plot must be drawn using the axis plot() function:

```
# generate angles from the radii
rs = linspace(0, 2, num=100)
thetas = [ 2*math.pi*r for r in rs]

ax = plt.subplot(projection='polar')
ax.plot(thetas, rs)
```

Listing F.14. Part of spiral.py

F.4.4 The Contour Lines Plot. A contour plot is our first 3D graph, albeit one using a 2D mapping. It's a member of matplotlib's "Gridded Data" class of plots, which also include filled contour plots and quiver charts (`https://matplotlib.org/stable/plot_types/arrays/`).

Listing F.15 (contourLines.py) plots the difference of two 2D Gaussian functions, with labels drawn on their contour lines (see Fig. F.14).

```
y = np.arange(-2.0, 2.0, 0.025)
X, Y = np.meshgrid(x, y)   # numpy output required

# build curve for generating z-coords
Z1 = np.exp(-X**2 - Y**2)  # 2D Gaussian
Z2 = np.exp(-(X-1)**2 - (Y -1)**2)
Z = (Z1 - Z2) * 2  # difference of Gaussians

cts = plt.contour(X, Y, Z)   # , colors="k")
plt.clabel(cts, inline=True, fontsize=10)
plt.title('Labeled Contour Lines')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Listing F.15. Labeled contour lines representing a 3D equation

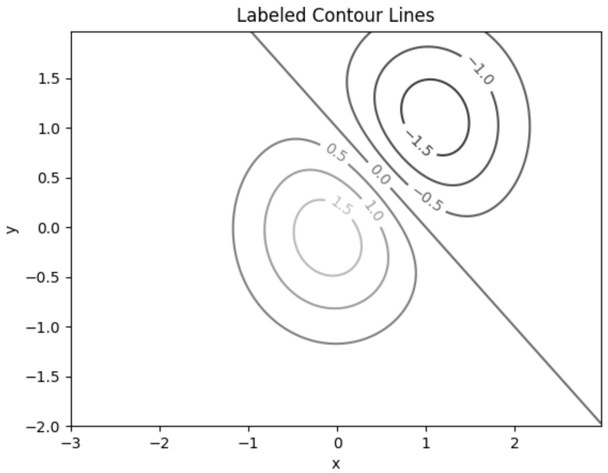

Figure F.14. A contour lines plot

A useful variant is to add the argument `colors="k"` to contour(), which draws the contours in black, with negatives represented by dashed lines.

The definition of the equation requires that the x- and y- coordinates be organized into a numpy meshgrid, which changes the x- and y- axis data into 2D arrays; the essential idea is shown in Fig. F.15.

This permits the equation to be written very succinctly since the X and Y numpy arrays are automatically combined element-by element. The corresponding equation is:

$$z = 2 \times (\exp^{(-x^2-y^2)} - \exp^{(-(x-1)^2-(y-1)^2)})$$

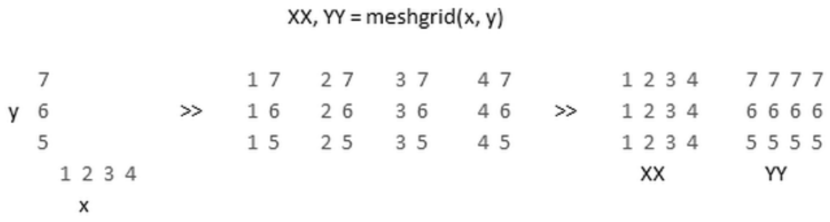

Figure F.15. Converting data into a numpy meshgrid

F.4.5 The Heat Map. A heat map is a way of displaying 3D data in a 2D form when the z- values can be stored as a 2D array or list.

Listing F.16 (colorMap.py) produces a *categorical* heat map (Fig. F.16) since the x- and y- data are categories (in this case, student names and subjects).

```
nms = ['Sam','Jim','John','Sue','Mike','Jane']
subjs = ['Maths','Physics','English','Chemistry',
         'History','Comp Sci']
marks = [[50, 74, 40, 59,90, 98],
         [72, 85, 64, 33, 47, 87],
         [52, 97, 44, 73, 17, 56],
         [69, 45, 89, 79,70, 48],
         [87, 65, 56, 86, 72, 68],
         [90, 29, 78, 66, 50, 32]]

# display the categories as x- and y- axis labels
plt.xticks(ticks=range(len(nms)), labels=nms, rotation=90)
plt.yticks(ticks=range(len(subjs)),labels=subjs)

hm=plt.imshow(marks,cmap='Blues_r',interpolation="nearest")
plt.colorbar(hm)
plt.title("Students and their Subject Marks")
plt.show()
```

Listing F.16. A blue color map relating students, subjects, and their scores

The map is displayed using imshow() which draws a 2D list (or numpy array) as a colored checkboard. In fact, imshow() can also be passed a RGB image for rendering (as the function name suggests).

Fig. F.16 is colored using a reversed version of one of matplotlib's sequential colormaps, 'Blues'. Other map categories include diverging, cyclic, and qualitative; for details see `https://matplotlib.org/stable/users/explain/colors/colormaps.html`. It also includes a color bar, which explains how colors are assigned to values.

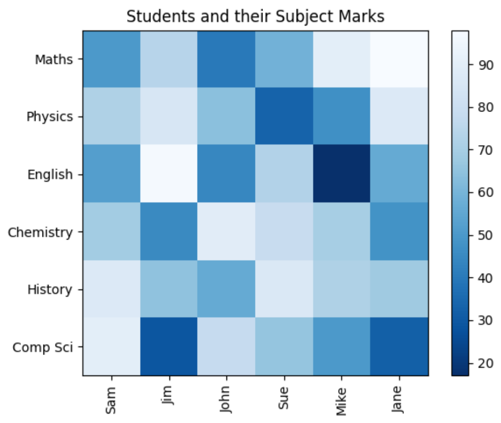

Figure F.16. A heat map of student marks

F.5 3D Plots

This section describes four types of 3D charts: line, scatter, wireframe, and surface plots. Several more can be found at `https://matplotlib.org/stable/plot_types/3D/`, and there are numerous examples at `https://matplotlib.org/stable/gallery/mplot3d/`.

One very nice feature common to all 3D plots is that matplotlib's window controls allow a figure to be rotated in addition to being panned and zoomed.

F.5.1 3D Lines or Points. The generation of 3D line charts and scatter plots is so similar that we've combined the examples into one program, Listing F.17 (points3D.py). If plot() is called then a line appears as in Fig. F.17(a), whereas scatter() generates Fig. F.17(b).

```
ax = plt.figure().add_subplot(projection ='3d')

# define data
zs = linspace(0, 1, 100)
xs = [ z * math.sin(25 * z) for z in zs]
ys = [ z * math.cos(25 * z) for z in zs]

plt.plot(xs, ys, zs)    # line
# plt.scatter(xs, ys, zs=zs, s=70)  # points; s is size

plt.title('3D plot')
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlabel('z')   # no plt.zlabel()
plt.show()
```

Listing F.17. Rendering 3D points

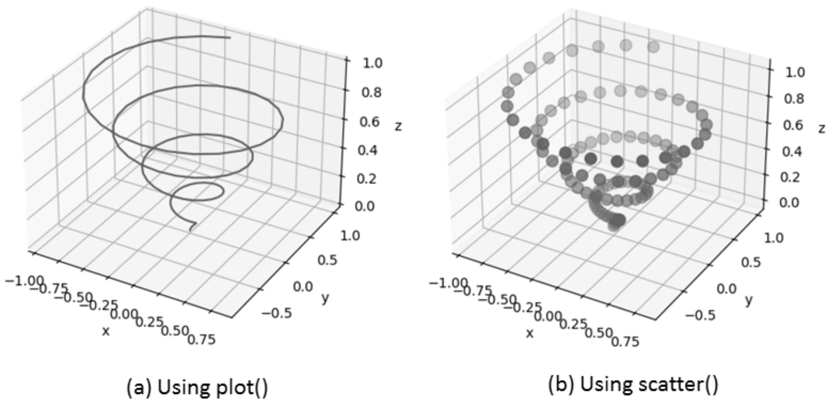

Figure F.17. 3D line or points

For plot() and scatter() to generate 3D plots rather than their usual 2D graphs, it's necessary to set the projection attribute to '3d'. Also, z-axis labeling must be set via an axis function.

F.5.2 A Wireframe Plot. Producing a wireframe, and the surface plot of the next section, requires that the x- and y- data be represented by a numpy meshgrid (which we met when drawing contour lines). Listing F.18 (wire3D.py) produces the graph in Fig. F.18.

```
X = np.linspace(-1, 2, 20)  # increase step for more detail
Y = np.linspace(-1, 2, 20)
X, Y = np.meshgrid(X, Y)  # numpy output required
Z = np.sin(np.sqrt(X**2 + Y**2))

ax = plt.axes(projection = '3d')
ax.plot_wireframe(X, Y, Z)
plt.title('3D Wireframe Plot')
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlabel('z')  # no plt.zlabel()
plt.show()
```

Listing F.18. A wireframe plot of a 3D equation

The distance between the wires is controlled by the intervals between the x- and y- values. It's also possible to set the spacing via 'count' or 'stride' arguments to plot_wireframe().

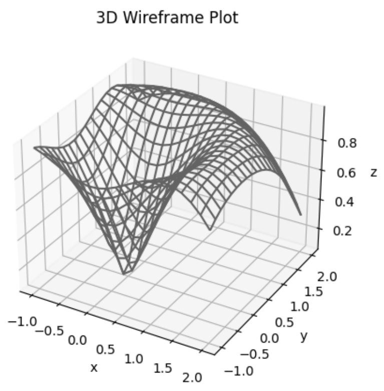

Figure F.18. A wireframe

F.5.3 A Surface Plot. A surface plot can be thought of as a filled and colored wireframe, and its equation uses a meshgrid in the same way. The differences come in the optional coloring arguments to plot_surface(). The simplest approach is to use a predefined colormap (which we met when using heat maps). It's also a good idea to adjust the color and thickness of the wires.

Listing F.19 (surface3D.py) produces the graph in Fig. F.19.

```
x = np.linspace(-10, 10, 40)
y = np.linspace(-10, 10, 40)
X, Y = np.meshgrid(x, y)    # numpy output required
Z = fn(X, Y)

ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z,
    cmap="Blues_r", edgecolor="Black", lw=0.2)

plt.title('Surface Plot of fn'+str(n))
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlabel('z')  # no plt.zlabel()
plt.show()
```

Listing F.19. Generate a 3D mesh of a function

We haven't shown all of the code in Listing F.19. There are seven equations defined as Python functions, and the user must input a number to select a function to generate the surface. Function no. 6 shown in Fig. F.19 is:

```
def fn6(x, y):   # monkey saddle
  r = np.sqrt(x*x + y*y)
  return x*y*(x-y)*(x+y)/r
```

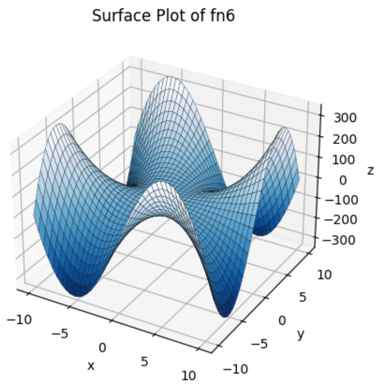

Figure F.19. A surface plot of a monkey saddle

F.6 Animation

animBall.py utilizes matplotlib's FuncAnimation() to animate the trajectory of a ball as a series of points plotted over time. Fig. F.20 shows two frames from the animation.

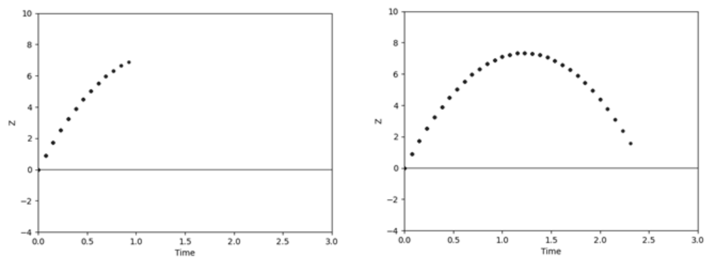

Figure F.20. Two frames from the ball animation

The important code from animBall.py:

```
ts = linspace(0, 3, 40)
g = -9.81
v0 = 12
zs = [((g*t**2)/2 + v0*t) for t in ts]

fig, ax = plt.subplots()
anim = animation.FuncAnimation(fig, update,
               frames=len(ts)-1, interval=50) # ms
```

After all of the points have been generated, FuncAnimation() automatically calls update() every 50 ms. The function only draws *some* of those points depending on the current animation frame counter.

```
def update(i):
  if i == 0:  # reset for next play
    ax.clear()
    ax.set(xlim=[0, 3], ylim=[-4, 10],
           xlabel='Time', ylabel='Z')
    ax.axhline(y=0, color='k', alpha=0.5)
  ax.scatter(ts[:i], zs[:i], c="b", s=10)
```

The current frame number, *i*, is used on the last line to call scatter() with the data sliced to only include the first *i* points.

Although the function called by FuncAnimation() can have any name, it can only receive the frame number as an input argument. That means that the rest of the plotting data (e.g. the `ts` and `zs` lists, and `ax`) must be globals.

An extra feature of animBall.py is the ability for the user to pause/resume the animation by clicking anywhere inside the window. This is implemented by attaching a listener to button press events:

```
fig.canvas.mpl_connect('button_press_event', onClick)
animRunning = True
```

When a press event is detected, onClick() is called:

```
def onClick(event):
  global animRunning
  if animRunning:
    anim.event_source.stop()
    animRunning = False
  else:
    anim.event_source.start()
    animRunning = True
```

As with update(), the function can be called anything but can only accept an event value. As a consequence, the `animRunning` boolean must be declared global since its value is toggled inside the function. Event processing is disabled or re-enabled, which stops or restarts FuncAnimation().

The matplotlib user guide contains a good description of its animation features (`https://matplotlib.org/stable/users/explain/animations/animations.html#animations`), and the examples page has a section dedicated to animations (`https://matplotlib.org/stable/gallery/index.html#animation`). The button press functionality is part of matplotlib's event handling features, which has its own section on the examples page (`https://matplotlib.org/stable/gallery/event_handling/`).

F.7 Widgets

Widgets are GUI controls, such as sliders and buttons, which can be used to control and modify a plot interactively. These controls work in a similar way to FuncAnimation() (see the previous section), but instead of responding to time events they're triggered by the user clicking on the control; a function is then called to modify the plot.

Listing F.20 (sineSlider.py) draws the sine wave shown in Fig. F.21, and allows its frequency to be adjusted via a slider.

```
def update(value):
  ys = [ math.sin(value * math.pi * x) for x in thetas]
  line.set_ydata(ys)
  fig.canvas.draw_idle()

fig, ax = plt.subplots()
plt.subplots_adjust(bottom=0.2)
thetas = linspace(0, math.pi, 300)
ys = [ math.sin(5 * math.pi * x) for x in thetas]
line, = ax.plot(thetas, ys)

# frequency slider
freqAxis = plt.axes([0.25, 0.1, 0.65, 0.03])
freqSlider = Slider(freqAxis, 'Frequency [Hz]',
                            0, 10, valinit=5)
                        # min, max, initial value
freqSlider.on_changed(update)
plt.show()
```

Listing F.20. Using a slider to control the frequency of a sine wave

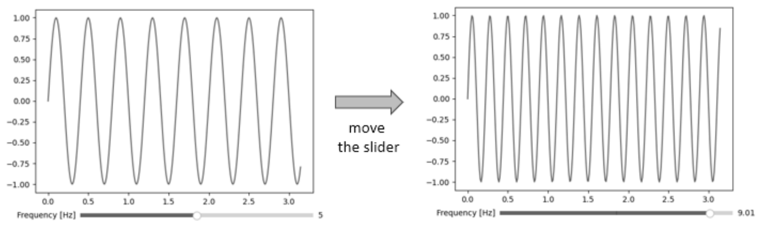

Figure F.21. Changing sine wave frequency

Setting up a control typically takes three lines of code: a line to specify the widget size in terms of axis values, the creation of the control, and the linking of the control to a function.

The matplotlib API documentation has a detailed description of widget features (`https://matplotlib.org/stable/api/widgets_api.html`), and the examples page has a dedicated section on the topic (`https://matplotlib.org/stable/gallery/widgets/`).