## 3.4 Introducing Graphics

SY v1

The graphical presentation of numerical data is perhaps the most important output of an engineering computing program. We must begin by understanding the purpose of graphics:

Graphics *reveal* data. (Tufte 2001; p. 13)

Data can, of course, be presented in other ways. Small sets of data are sometimes best presented in table format. However, most data is best presented visually.

Good graphics require careful design. The following list characterizes some aspects of a quality graphic (p. 13):

- It shows the data
- It draws the viewer to the data, not its presentation
- It presents the truth of the data with minimal distortion
- It presents lots of data in a small space
- It makes understandable large data sets
- It draws the viewer to compare pieces of the data set
- It presents the data in important levels of detail (broad and fine)
- It has a clear purpose: description, exploration, tabulation, or decoration
- It is closely integrated with accompanying descriptions of the data set

A good graphic is an explanation. For instance, it explains how one variable is related to another. In some cases, a causal explanation is suggested, as in figure 3.2, which suggests economic elites have much greater influence on policy adoption than do average citizens.

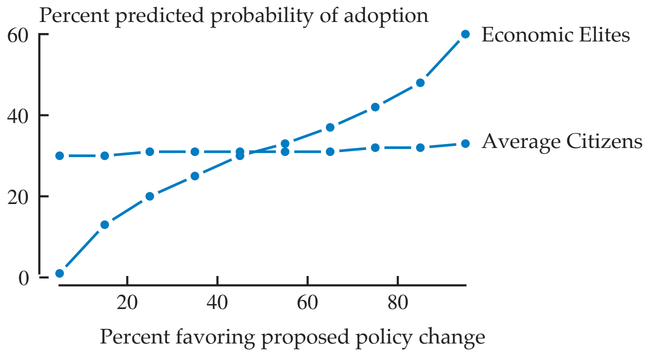

Figure 3.2. Percent predicted probability of public policy adoption for economic elites and average citizens. Study, results, and statistical model by Gilens and Page (2014).

A powerful Python package for creating graphics is Matplotlib (Hunter 2007). It is included in the base Anaconda environment and its most important module can be loaded with the following statement:

```
import matplotlib.pyplot as plt
```

In this book, we will use `plt` as the name for this module.

The rest of this section introduces the three fundamental types of graphics: function graphs, plots, and charts. Several important Matplotlib functions and methods are presented for generating each fundamental type of graphic.

### 3.4.1 Function Graphs

A **function graph** is a graphic that displays the relationship between a function and one or more of its arguments. A single 2D function graph can display the relationship between a function and a single argument. For instance, consider the polynomial function

$$f(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0,$$

for real constant coefficients $a_0, \cdots, a_4$. To visualize the function for a given set of coefficients,

$$a_0, \cdots, a_4 = 10, 10, -20, -1, 1,$$

we could write a Python program that begins by defining the function $f$ as a Python function as follows:

```
def f(x):
    a0, a1, a2, a3, a4 = 10, 10, -20, -1, 1
    return a4 * x ** 4 + a3 * x ** 3 + a2 * x ** 2 + a1 * x + a0
```

Our strategy is to create two NumPy arrays, one for values of $x$ and another for corresponding values of $y = f(x)$. These values should cover the domain of interest; for instance,

```
x = np.linspace(-5, 5, 101)
y = f(x)
```

The following code will create a figure and an axis, plot `x` and `y` on the axis, set the $x$-axis and $y$-axis labels, and display the figure:

```
fig, ax = plt.subplots()  # Create a figure and an axis
ax.plot(x, y)
ax.set_xlabel("x")  # Label the x axis
ax.set_ylabel("f(x)")  # Label the y axis
plt.show()  # Display the figure
```

Consider each of the lines and what it does:

- `fig, ax = plt.subplots()` This function returns two objects with fundamental Matplotlib classes: the **figure** class `matplotlib.figure.Figure` and the **axes** class `matplotlib.axes.Axes`. Figures are the top-level containers for all elements in a Matplotlib graphic. Axes are the containers for individual plots and subplots, multiple of which can be contained in a single figure.
- `ax.plot(x, y)` This axes method plots `y` versus `x`, drawing a continuous curve connecting the points $(x_i, y_i)$. There are many optional arguments we will later explore, but the defaults will suffice for this example.
- `ax.set_xlabel("x")` This axes method sets the $x$-axis label to the string argument.
- `ax.set_ylabel("f(x)")` This sets the $y$-axis label.
- `plt.show()` This function displays all open figures. In an IPython session (e.g., one in Spyder), this is superfluous because figures are automatically displayed in this environment.

The execution of this code displays a figure similar to the stylized version shown in figure 3.3.[4] Note that although Matplotlib connects the individual points $(x_i, y_i)$ on the curve with straight lines, with enough points the curve appears smooth.

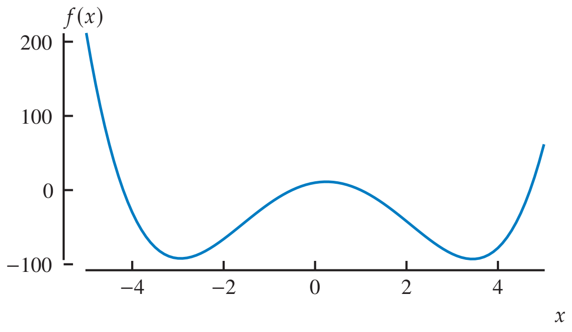

Figure 3.3. A graph of polynomial $f(x)$.

A Matplotlib figure can contain multiple axes objects and each axes object can contain multiple plots. We will explore the former in a later section and the latter in section 3.4.2.

4. We will later explore how to style and save figures. The stylization of book figures will be minimal but necessary to demonstate the aesthetic cohesion with the text for which we should strive.

### 3.4.2 Plots

A **plot** is a graphic that displays discrete data in relation to one or more coordinates in a coordinate system. Consider a **data set**, a set of **data points**, $n$-tuples $(x_{0i}, \cdots, x_{ni})$, in a **coordinate system** $(x_0, \cdots, x_n)$. A plot of the data set would display each of the data points in the data set. For instance, a plot of a data set in a Cartesian coordinate system $(x, y)$ would display each of the data points $(x_i, y_i)$ in the data set.

You may have observed that to create a function graph in section 3.4.1 we generated a data set of Cartesian data points $(x_i, y_i)$ and actually created a *plot* of that data set. It turns out that a function graph is really just a plot that tries to minimize the appearance of individual data points to emphasize the continuously varying nature of the function it is presenting. On the other hand, plots that are not function graphs should usually emphasize its data points.

Experimental data are frequently presented in plots. For example, consider a data set collected in an experiment exploring the relationship among the pressure $P$, volume $V$, and temperature $T$ of a noble gas. You may recall that noble gases are good approximations of an ideal gas, which obeys the ideal gas law

$$PV = nRT,$$

where $n$ is the (molar) amount of gas and $R$ is the ideal gas constant (approximately 8.314 J/(K·mol)). Our Engcom package `data` module simulates this data set; the module can be imported with the following statement:

```python
import engcom.data
```

A data set can be generated for values of volume `V` and temperature `T` with the following function call:

```python
d = engcom.data.ideal_gas(
    V=np.linspace(1, 2, 16),  # (m^3) Volume values
    T=np.linspace(273, 573, 4),  # (K) Temperature values
)
```

Now `d` is a dictionary with the following key–value pairs:

- `"volume"`–$V_{16\times1}$ (m$^3$)
- `"temperature"`–$T_{1\times4}$ (K)
- `"pressure"`–$P_{16\times4}$ (Pa)

We would like to plot $P$ versus $V$ for each of the 4 temperatures $T$; that is, plot a sequence of pairs $(P_i, V_i)$ for each $T_j$. The following code loops through the temperatures and plots to the same axes object:

```
fig, ax = plt.subplots()
for j, Tj in enumerate(d["temperature"].flatten()):
    x = d["volume"]  # (m^3)
    y = d["pressure"][:,j] / 1e6  # (MPa)
    ax.plot(x, y, marker="o", color="dodgerblue")  # Circle markers
    ax.text(x=x[-1], y=y[-1], s=f"$T = {Tj}$ K")  # Label last point
```

Finally, we label the axes and display the figure with the following code:

```
ax.set_xlabel("Volume (m$^3$)")
ax.set_ylabel("Pressure (MPa)")
plt.show()
```

The figure should appear as shown in figure 3.4.

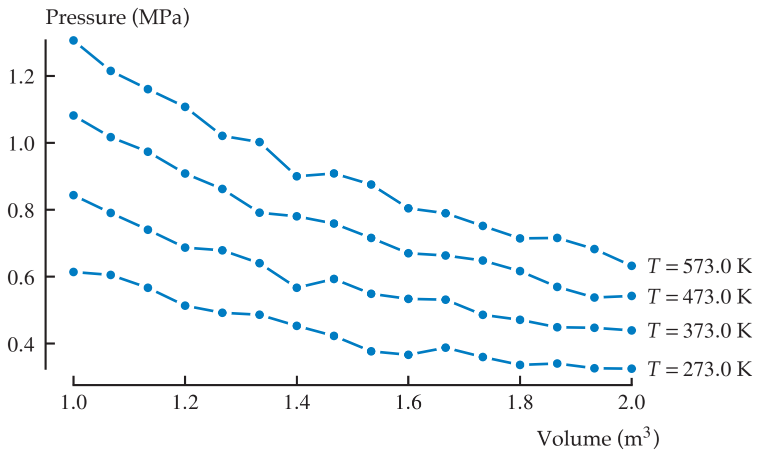

Figure 3.4. Ideal gas pressure versus volume for different temperatures.

Note the practice of labeling the individual data plots instead of creating a separate legend. At times a legend is necessary, but often it is better to simply label the data so the viewer needn't perform unnecessary work moving back-and-forth between the legend and the plot.

### 3.4.3 Charts

The third fundamental type of graphic is the **chart**: a data set presentation in which **signs** (i.e., icons, indices, and symbols)[5] represent data. Some signs have become commonplace for representing data:

- The **dot** • represents a data point
- The **curve** ∼ represents a continuously varying quantity or a connection between sequential data points
- The **bar** ▬ represents a quantity via its length

A function graph (section 3.4.1) represents a continuously varying quantity with a curve. A plot (section 3.4.2) represents data points with dots and connections among sequential data points with curves. The chart can use dots, curves, bars, or any other sign to represent data. Therefore, "chart" is the most general term: a function graph is a type of plot, which is a type of chart.

There are many flavors of chart in addition to the function graph and plot. Perhaps the most important are variations on the bar chart and the related histogram, covered here.

**3.4.3.1 Bar Charts** A **bar chart** represents and compares quantities of some type (e.g., density) for a collection of discrete **categories** (e.g., liquids). The categories may have a natural progression, in which case they should be ordered accordingly; otherwise, they should be ordered by quantity.

Consider the bar chart of thermal conductivity for various metals shown in figure 3.5. The quantity charted is thermal conductivity and the categories are types of metal. Note that not only can we easily see the conductivity of each metal, we can easily compare conductivities in this graphic. A simple table of data would be much less informative in this regard.

5. The field of **semiotics** (the study of signs) defines a sign as something that communicates a meaning. Charles Sanders Peirce distinguished three types of signs in terms of a sign's relation to its object: an **icon** has a topographical similarity with its object (e.g., ☽ is an icon representing a waxing moon), an **index** indicates something else (e.g., ↑ points to something), and a **symbol** is a sign for an object only by convention (e.g., ☣ is the biohazard symbol). The type of a given sign can be ambiguous (e.g., ☜ is an icon insofar as its object is a hand, but is an index insofar as it indicates the direction *left*).

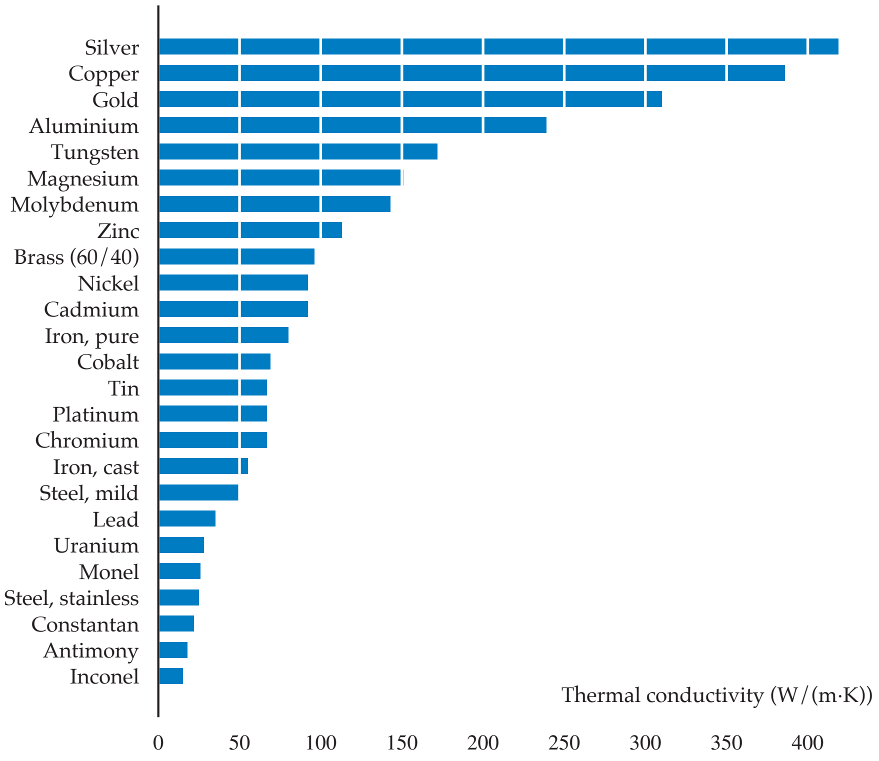

Figure 3.5. A bar chart of thermal conductivity for metals (data from Carvill (1994)).

We can produce the bar chart of figure 3.5 as follows. Begin by loading packages:

```python
import numpy as np
import matplotlib.pyplot as plt
import engcom.data
```

The data can be loaded from the `engcom.data` module as follows:

```python
d = engcom.data.thermal_conductivity(category="Metals", paired=False)
y = np.arange(len(d["labels"]))
x_alpha = d["conductivity"]  # Alphabetically sorted
labels_alpha = np.array(d["labels"])  # Alphabetically sorted
```

The data is here sorted alphabetically. However, we prefer to sort it by quantity, which can be achieved with the use of the `np.lexsort()` function as follows:

```
ix = np.lexsort((labels_alpha, x_alpha))  # Sort indices
x = x_alpha[ix]
labels = labels_alpha[ix].tolist()
```

Now we can use Matplotlib's `ax.barh()` axes method (for a vertical bar chart, use `ax.bar()`) as follows:

```
fig, ax = plt.subplots()
ax.barh(y, x, color="dodgerblue")
ax.set_yticks(y, labels=labels)
ax.set_xlabel("Thermal conductivity (W/(m$\\cdot$K))")
```

In some cases we present a group of subcategories for each category, in which case a `tuple` of subcategories can be passed to `ax.barh()` or `ax.bar()`.

There are other ways to present this type of information (i.e., quantities for a collection of categories), but it is difficult to do better than the bar chart.

**3.4.3.2 Histograms** A **histogram** is a chart that presents a distribution of a variable. Its use of bars makes it closely related to the bar chart, but it represents the frequency a variable falls in each **bin**: an interval of values treated as a category.

Consider the histogram of my movie ratings on a 0–10 scale shown in figure 3.6. Ratings most frequently fall in the $[6, 7)$ bin. Only two movies are in the $[9, 10]$ bin. With the histogram we can easily compare the relative frequencies of values.

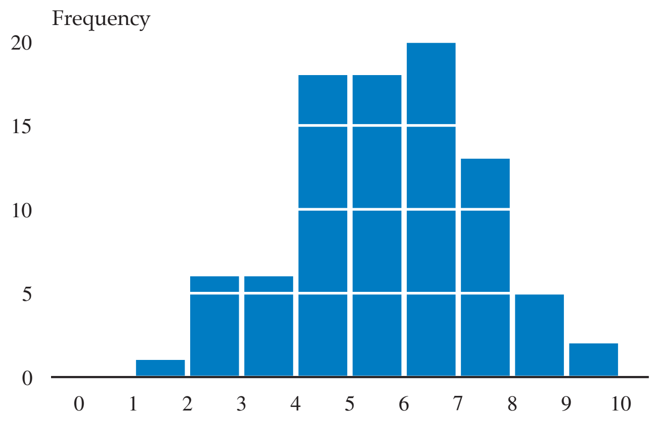

Figure 3.6. A histogram of my movie ratings on a 0–10 scale.

We can produce the histogram chart of figure 3.6 as follows. After loading the same packages as we did for the bar chart, the data can be loaded from the `engcom.data` module as follows:

```python
d = engcom.data.movie_ratings_binned()
x = list(range(0,len(d["rating_freq"])))
```

Now we can use Matplotlib's `ax.bar()` axes method (for a horizontal histogram, use `ax.barh()`) as follows:

```python
fig, ax = plt.subplots()
ax.bar(x, d["rating_freq"], color="dodgerblue", width=.9)
ax.set_xticks(x)
ax.set_xticklabels(d["labels"])
ax.set_xlabel("Rating out of $10$")
ax.set_ylabel("Frequency")
```

Note that Matplotlib does have a `hist()` function that can make generating histograms slightly easier. However, we prefer the flexibility of the `bar()` method.

#### 3.4.3.3 Other Types of Charts

## 3.5 Problems

A5

**Problem 3.1** J3 Write a program that meets the following requirements:

a. It constructs a NumPy matrix `A` to represent the following mathematical matrix:
$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \end{bmatrix}.$$
b. It defines a function `left_up_sum(A: np.ndarray) -> np.ndarray` that adds the component (element) to the left and the component above (wrapping, if necessary) to each component. The function should pass through the array once, row-by-row, and return a new array. The function should be able to handle any size of matrix.
c. It defines a function `left_up_sums(A: np.ndarray, n: int) -> np.ndarray` that executes `left_up_sum()` `n` times and returns a new array.
d. It calls `left_up_sums()` on `A` and prints the returned array for the following values of `n`: `0`, `1`, `4`.

**Problem 3.2** ZF The inner product of two real $n$-vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is defined as
$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{i=0}^{n-1} x_i y_i.$$
The result is a scalar. The `np.inner()` and `np.dot()` functions can be used in NumPy to find the inner product of two vectors of the same size. In this problem, we will write our own function that computes the real inner product even if they are of different sizes. Write a program that meets the following requirements:

a. It defines a function
```
inner_flat_trunc(x: np.ndarray, y: np.ndarray) -> float
```
that returns the truncated inner product of vectors `a` and `b` even if the sizes of the vectors do not match by using a truncated version of the one that is too long. The function should handle any shape of input arrays by using the `flatten()` method before truncating and taking the inner product. If both input arrays do not have `dtype` attribute `np.dtype('float')` or `np.dtype('int')`, the function should raise a `TypeError` exception.
b. It calls `inner_flat_trunc()` on the following arrays:
   i. A pair of arrays from the lists:
   `[-1.1, 3, 2.9, -1, -9.2, 0.1]` and `[1.3, 0.2, 8.3]`