3 Interlude: simple plotting with `pylab`

As Python has grown in popularity, many libraries of packages and modules have become available to extend its functionality in useful ways; Matplotlib is one such library. Matplotlib provides a means of producing graphical plots that can be embedded into applications, displayed on the screen or output as high-quality image files for publication.

Matplotlib has a fully fledged *object-oriented* interface, which is described in more detail in Chapter 7, but for simple plotting in an interactive shell session, its simpler, *procedural* `pylab` interface provides a convenient way of visualizing data. `pylab` is designed to be easy to learn and functions in a similar way to comparable tools in the commercial MATLAB package.

On a system with Matplotlib installed the `pylab` package is imported with

```
>>> import pylab
```

even though this means prefacing all of the `pylab` method calls with "`pylab.`"[1]

3.1 Basic plotting

3.1.1 Line plots and scatterplots

The simplest (x, y) line plot is achieved by calling `pylab.plot` with two iterable objects of the same length (typically lists of numbers or NumPy arrays). For example,

```
>>> ax = [0., 0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
>>> ay = [0.0, 0.25, 1.0, 2.25, 4.0, 6.25, 9.0]
>>> pylab.plot(ax,ay)
>>> pylab.show()
```

`pylab.plot` creates a matplotlib object (here, a `Line2D` object) and `pylab.show()` displays it on the screen. Figure 3.1 shows the result; by default the line will be in blue.

To plot (x, y) points as a scatterplot rather than as a line plot, call `pylab.scatter` instead:

```
>>> import random
>>> ax, ay = [], []
```

[1] It is better to avoid polluting the global namespace by importing as `from pylab import *`.

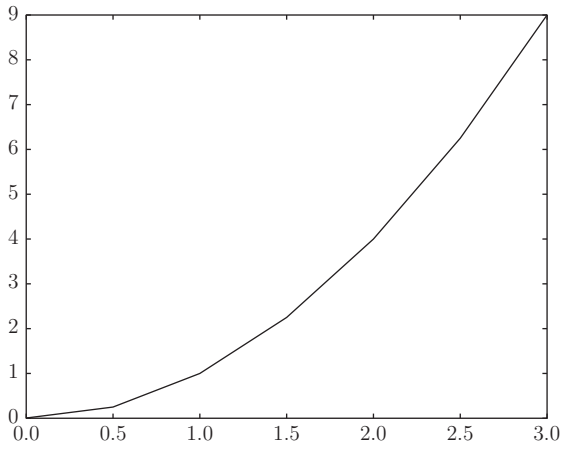

Figure 3.1 A basic (x, y) line plot.

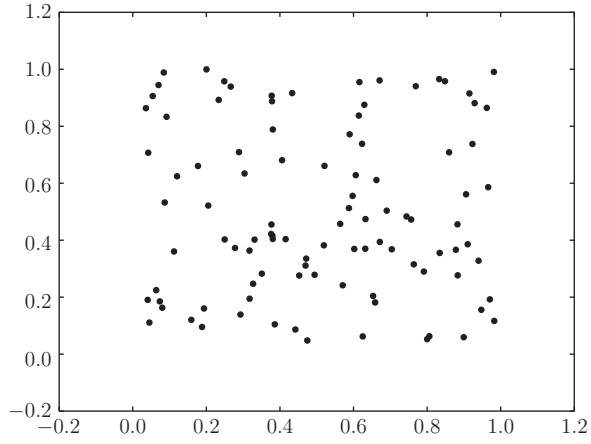

Figure 3.2 A basic scatter plot.

```
>>> for i in range(100):
...     ax.append(random.random())
...     ay.append(random.random())
...
>>> pylab.scatter(ax,ay)
>>> pylab.show()
```

The resulting plot is shown in Figure 3.2.

We can also save the plot as an image by calling `pylab.savefig(`*`filename`*`)`. The desired image format is deduced from the filename extension. For example,

```
pylab.savefig('plot.png')       # save as a PNG image
pylab.savefig('plot.pdf')       # save as PDF
pylab.savefig('plot.eps')       # save in Encapsulated PostScript format
```

Example E3.1 As an example, let's plot the function $y = \sin^2 x$ for $-2\pi \leq x \leq 2\pi$. Using only the Python we've covered in the previous chapter, here is one approach:

We calculate and plot 1,000 (x, y) points, and store them in the lists `ax` and `ay`. To set up the `ax` list as the abcissa, we can't use `range` directly because that method only produces integer sequences, so first we work out the spacing between each x value as

$$\Delta x = \frac{x_{\max} - x_{\min}}{n - 1}$$

(if our n values are to *include* $x_{\min}$ and $x_{\max}$, there are $n - 1$ intervals of width Δx); the abcissa points are then

$$x_i = x_{\min} + i\Delta x \quad \text{for } i = 0, 1, 2, \cdots, n - 1.$$

The corresponding y-axis points are

$$y_i = \sin^2(x_i).$$

The following program implements this approach, and plots the (x, y) points on a simple line-graph (see Figure 3.3).

Listing 3.1 Plotting $y = \sin^2 x$

```
# eg3-sin2x.py

import math
import pylab
xmin, xmax = -2. * math.pi, 2. * math.pi
n = 1000
x = [0.] * n
y = [0.] * n
dx = (xmax - xmin)/(n-1)
for i in range(n):
    xpt = xmin + i * dx
    x[i] = xpt
    y[i] = math.sin(xpt)**2

pylab.plot(x,y)
pylab.show()
```

3.1.2 `linspace` and vectorization

Plotting the simple function $y = \sin^2 x$ in the previous example involved quite a lot of work, almost all of it to do with setting up the lists `x` and `y`. In fact, `pylab` provides some of the same functionality as the NumPy library introduced in Chapter 6, which can be used to make life much easier.

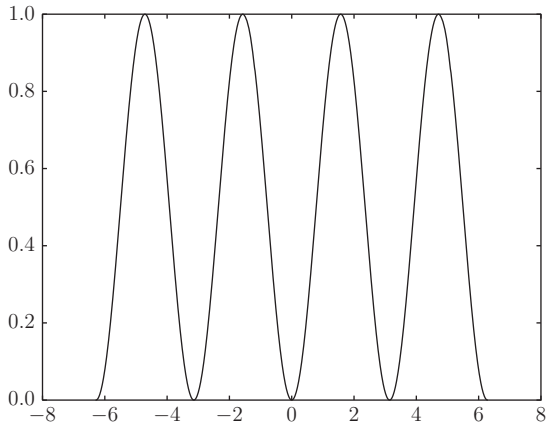

Figure 3.3 A plot of $y = \sin^2 x$.

First, the regularly spaced grid of x-coordinates, `x`, can be created using `linspace`. This is much like a floating point version of the `range` built-in: it takes a start value, an end value, and the number of values in the sequence and generates an array of values representing the arithmetic progression between (and *inclusive of*) the two values. For example, `x = pylab.linspace(−5, 5, 1001)` creates the sequence: $-5.0, -4.99, -4.98, ..., 4.99, 5.0$.

Second, the `pylab` equivalents of the `math` module's methods can act on iterable objects (such as `lists` or NumPy arrays). Thus, `y = pylab.sin(x)` creates a sequence of values (actually, a NumPy `ndarray`), which are $\sin(x_i)$ for each value x_i in the array `x`:

```
import pylab
n = 1000
xmin, xmax = -2. * math.pi, 2. * math.pi
x = pylab.linspace(xmin, xmax, n)
y = pylab.sin(x)**2
pylab.plot(x,y)
pylab.show()
```

This is called *vectorization* and is described in more detail in Section 6.1.3. Lists and tuples can be turned into array objects supporting vectorization with the `array` constructor method:

```
>>> w = [1.0, 2.0, 3.0, 4.0]
>>> w = pylab.array(w)
>>> w * 100     # multiply each element by 100
array([ 100.,  200.,  300.,  400.])
```

To add a second line to the plot, simply call `pylab.plot` again:

```
...
x = pylab.linspace(xmin, xmax, n)
y1 = pylab.sin(x)**2
```

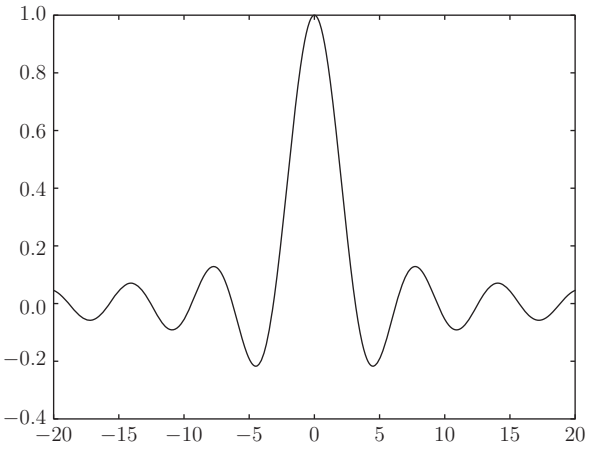

Figure 3.4 A plot of $y = \mathrm{sinc}(x)$.

```
y2 = pylab.cos(x)**2
pylab.plot(x,y1)
pylab.plot(x,y2)
pylab.show()
```

Note that after a plot has been displayed with `show` or saved with `savefig`, it is no longer available to display a second time – to do this it is necessary to call `pylab.plot` again. This is because of the procedural nature of the `pylab` interface: each call to a `pylab` method changes the internal *state* of the plot object. The plot object is built up by successive calls to such methods (adding lines, legends and labels, setting the axis limits, etc.), and then the plot object is displayed or saved.

Example E3.2 The sinc function is the function

$$f(x) = \frac{\sin x}{x}.$$

To plot it over $20 \leq x \leq 20$:

```
>>> x = pylab.linspace(-20, 20, 1001)
>>> y = pylab.sin(x)/x

__main__:1: RuntimeWarning: invalid value encountered in true_divide
>>> pylab.plot(x,y)
>>> pylab.show()
```

Note that even though Python warns of the division by zero at $x = 0$, the function is plotted correctly: the singular point is set to the special value `nan` (standing for "not a number") and is omitted from the plot (Figure 3.4).

```
>>> y[498:503]
array([ 0.99893367,  0.99973335,         nan,  0.99973335,  0.99893367])
```

3.1.3 Exercises

Problems

P3.1.1 Plot the functions

$$f_1(x) = \ln\left(\frac{1}{\cos^2 x}\right) \text{ and}$$
$$f_2(x) = \ln\left(\frac{1}{\sin^2 x}\right).$$

on 1,000 points across the range $-20 \leq x \leq 20$. What happens to these functions at $x = n\pi/2$ ($n = 0, \pm 1, \pm 2, \ldots$)? What happens in your plot of them?

P3.1.2 The *Michaelis-Menten* equation models the kinetics of enzymatic reactions as

$$v = \frac{\mathrm{d[P]}}{\mathrm{d}t} = \frac{V_{\max}[\mathrm{S}]}{K_m + [\mathrm{S}]},$$

where v is the rate of the reaction converting the substrate, S, to product P, catalyzed by the enzyme. $V_{\max}$ is the maximum rate (when all the enzyme is bound to S) and the Michaelis constant, K_m, is the substrate concentration at which the reaction rate is at half its maximum value.

Plot v against [S] for a reaction with $K_m = 0.04$ M and $V_{\max} = 0.1\ \mathrm{M\,s^{-1}}$. Look ahead to the next section if you want to label the axes.

P3.1.3 The normalized Gaussian function centered at $x = 0$ is

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{x^2}{2\sigma^2}\right).$$

Plot and compare the shapes of these functions for standard deviations $\sigma = 1, 1.5$ and 2.

3.2 Labels, legends and customization

3.2.1 Labels and legends

Plot legend

Each line on a `pylab` plot can be given a label by passing a string object to its `label` argument. However, the label won't appear on the plot unless you also call `pylab.legend` to add a legend:

```
pylab.plot(ax, ay1, label='sin^2(x)')
pylab.legend()
pylab.show()
```

The location of the legend is, by default, the top right-hand corner of the plot but can be customized by setting the `loc` argument to the `legend` method to either the string or integer values given in Table 3.1.

Table 3.1 Legend location specifiers

String	Integer
`'best'`	0
`'upper right'`	1
`'upper left'`	2
`'lower left'`	3
`'lower right'`	4
`'right'`	5
`'center left'`	6
`'center right'`	7
`'lower center'`	8
`'upper center'`	9
`'center'`	10

The plot title axis labels

A plot can be given a title above the axes by calling `pylab.title` and passing the title as a string. Similarly, the methods `pylab.xlabel` and `pylab.ylabel` control the labeling of the *x*- and *y*-axes: just pass the label you want as a string to these methods. The optional additional attribute `fontsize` sets the font size in points. For example,

```
t = pylab.linspace(0., 0.1, 1000)
Vp_uk, Vp_us = 230, 110
f_uk, f_us = 50, 60
❶ V_uk = Vp_uk * pylab.sin(2 * pylab.pi * f_uk * t)
V_us = Vp_us * pylab.sin(2 * pylab.pi * f_us * t)
❷ pylab.plot(t*1000, V_uk, label='UK')
pylab.plot(t*1000, V_us, label='US')
pylab.title('A comparison of AC voltages in the UK and US')
pylab.xlabel('Time /ms', fontsize=16.)
pylab.ylabel('Voltage /V', fontsize=16.)
pylab.legend()
pylab.show()
```

❶ We calculate the voltage as a function of time (`t`, in seconds) in the United Kingdom and in the United States, which have different peak voltages (230 V and 110 V respectively) and different frequencies (50 Hz and 60 Hz).
❷ The time is plotted on the *x*-axis in milliseconds (`t*1000`) – see Figure 3.5.

Using LaTeX in `pylab`

You can use LaTeX markup in `pylab` plots, but this option needs to be enabled in Matplotlib's "rc settings," as follows:

```
pylab.rc('text', usetex=True)
```

Then simply pass the LaTeX markup as a string to any label you want displayed in this way. Remember to use raw strings (`r'xxx'`) to prevent Python from escaping any characters followed by LaTeX's backslashes (see Section 2.3.2).

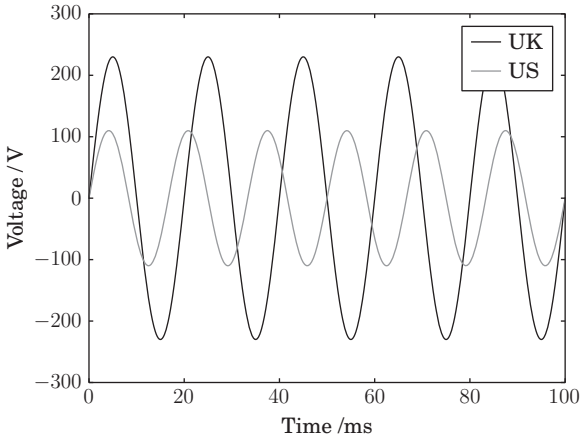

Figure 3.5 A comparison of AC voltages in the United Kingdom and United States.

Example E3.3 To plot the functions $f_n(x) = x^n \sin x$ for $n = 1, 2, 3, 4$:

```
import pylab
pylab.rc('text', usetex=True)

x = pylab.linspace(-10,10,1001)
for n in range(1,5):
    y = x**n * pylab.sin(x)
❶   y /= max(y)
    pylab.plot(x,y, label=r'$x^{}\sin x$'.format(n))
pylab.legend(loc='lower center')
pylab.show()
```

❶ To make the graphs easier to compare, they have been scaled to a maximum of 1 in the region considered.

The graph produced is given in Figure 3.6.

3.2.2 Customizing plots

Markers

By default, `plot` produces a line-graph with no markers at the plotted points. To add a marker on each point of the plotted data, use the `marker` argument. Several different markers are available and are documented online;[2] some of the more useful ones are listed in Table 3.2.

Colors

The color of a plotted line and/or its markers can be set with the `color` argument. Several formats for specifying the color are supported. First, there are one-letter codes

[2] http://matplotlib.org/api/markers_api.html#module-matplotlib.markers.

Table 3.2 Some Matplotlib marker styles

Code	Marker	Description
`.`	·	point
`o`	○	circle
`+`	+	plus
`x`	×	x
`D`	◇	diamond
`v`	▽	downward triangle
`^`	△	upward triangle
`s`	□	square
`*`	⋆	star

Table 3.3 Matplotlib color code letters

Code	Color
`b`	blue
`g`	green
`r`	red
`c`	cyan
`m`	magenta
`y`	yellow
`k`	black
`w`	white

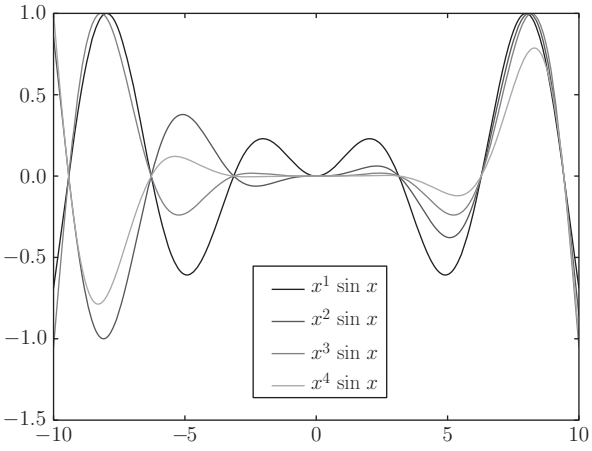

Figure 3.6 $f_n(x) = x^2 \sin x$ for $n = 1, 2, 3, 4$.

for some common colors, given in Table 3.3. For example, `color='r'` specifies a red line and markers. The default color sequence for a series of lines on the same plot is in the same order as this table.

Alternatively, shades of gray can specified as a string representing a `float` in the range 0–1 (`'0.'` being black and `'1.'` being white). HTML hex strings giving the

Table 3.4 Matplotlib line styles

Code	Line style
`-`	solid
`--`	dashed
`:`	dotted
`-.`	dash-dot

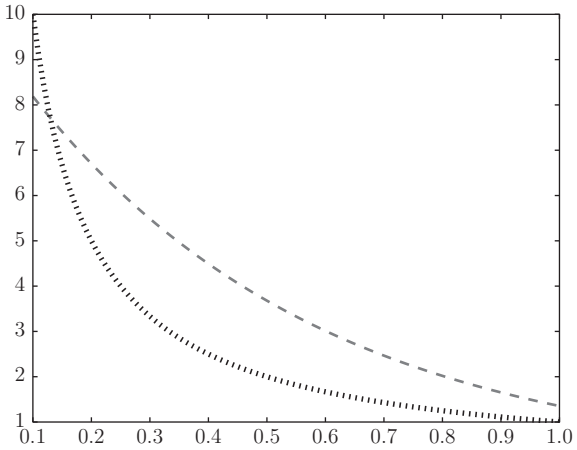

Figure 3.7 Two different line styles on the same plot.

red, green and blue (RGB) components of the color in the range `00` – `ff` can also be passed in the `color` argument (e.g., `color='#ff00ff'` is magenta). Finally, the RGB components can also be passed as a `tuple` of three values in the range 0–1 (e.g., `color=(0.5, 0., 0.)` is a dark red color).

Line styles and widths

The default plot line style is a solid line of weight 1.0 pt. To customize this, set the `linestyle` argument (also a string). Some of the possible line style settings are given in Table 3.4.

To draw no line at all, set `linestyle=''` (the empty string). The thickness of a line can be specified in points by passing a `float` to the `linewidth` attribute.

For example,

```
ax = pylab.linspace(0.1, 1., 100)
ayi = 1./ax
aye = 10. * pylab.exp(-2.*ax)
pylab.plot(ax, ayi, color='r', linestyle=':', linewidth=4.)
pylab.plot(ax, aye, color='m', linestyle='--', linewidth=2.)
pylab.show()
```

This code produces Figure 3.7.

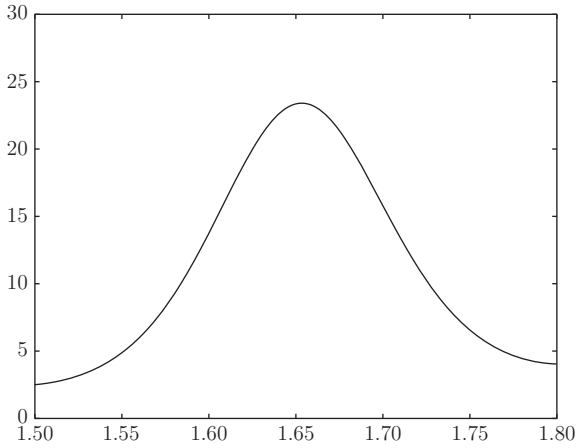

Figure 3.8 A plot produced with explicitly defined data limits.

The following abbreviations for the plot line properties are also valid:

- `c` for `color`
- `ls` for `linestyle`
- `lw` for `linewidth`

For example,

```
pylab.plot(x, y, c='g', ls='--', lw=2)      # a thick, green, dashed line
```

It is also possible to specify the color, linestyle and marker style in a single string:

```
pylab.plot(x, y, 'r:^')      # a red, dotted line with triangle markers
```

Finally, multiple lines can be plotted using a sequence of `x`, `y`, `format` arguments:

```
pylab.plot(x,y1, 'r--', x, y2, 'k-.')
```

plots a red dashed line for `(x,y1)` and a black dash-dot line for `(x,y2)`.

Plot limits

The methods `pylab.xlim` and `pylab.ylim` set the *x*- and *y*- limits of the plot respectively. They must be called *after* any `pylab.plot` statements, before showing or saving the figure. For example, the following code produces a plot of the provided data series between chosen limits (Figure 3.8):

```
t = pylab.linspace(0, 2, 1000)
f = t * pylab.exp(t + pylab.sin(20*t))
pylab.plot(t, f)
pylab.xlim(1.5,1.8)
pylab.ylim(0,30)
pylab.show()
```

Example E3.4 *Moore's Law* is the observation that the number of transistors on CPUs approximately doubles every two years. The following program illustrates this with

a comparison between the actual number of transistors on high-end CPUs from between 1972 and 2012, and that predicted by Moore's Law which may be stated mathematically as:

$$n_i = n_0 2^{(y_i - y_0)/T_2},$$

where n_0 is the number of transistors in some reference year, y_0, and $T_2 = 2$ is the number of years taken to double this number. Because the data cover 40 years, the values of n_i span many orders of magnitude, and it is convenient to apply Moore's Law to its logarithm, which shows a linear dependence on y:

$$\log_{10} n_i = \log_{10} n_0 + \frac{y_i - y_0}{T_2} \log_{10} 2.$$

Listing 3.2 An illustration of Moore's Law

```
# eg3-moore.py
import pylab

# The data - lists of years:
year = [1972, 1974, 1978, 1982, 1985, 1989, 1993, 1997, 1999, 2000, 2003,
        2004, 2007, 2008, 2012]
# and number of transistors (ntrans) on CPUs in millions:
ntrans = [0.0025, 0.005, 0.029, 0.12, 0.275, 1.18, 3.1, 7.5, 24.0, 42.0,
          220.0, 592.0, 1720.0, 2046.0, 3100.0]
# turn the ntrans list into a pylab array and multiply by 1 million
ntrans = pylab.array(ntrans) * 1.e6

y0, n0 = year[0], ntrans[0]
# A linear array of years spanning the data's years
y = pylab.linspace(y0, year[-1], year[-1] - y0 + 1)
# Time taken in years for the number of transistors to double
T2 = 2.
moore = pylab.log10(n0) + (y - y0) / T2 * pylab.log10(2)

pylab.plot(year, pylab.log10(ntrans), '*', markersize=12, color='r',
           markeredgecolor='r', label='observed')
pylab.plot(y, moore, linewidth=2, color='k', linestyle='--',
           label='predicted')
pylab.legend(fontsize=16, loc='upper left')
pylab.xlabel('Year', fontsize=16)
pylab.ylabel('log(ntrans)', fontsize=16)
pylab.title("Moore's Law")
pylab.show()
```

In this example, the data are given in two lists of equal length representing the year and representative number of transistors on a CPU in that year. The Moore's Law formula above is implemented in logarithmic form, using an array of years spanning the provided data. (Actually, since on a logarithmic scale this will be a straight line, really only two points are needed.)

For the plot, shown in Figure 3.9, the data are plotted as largeish stars and the Moore's Law prediction as a thick black line.

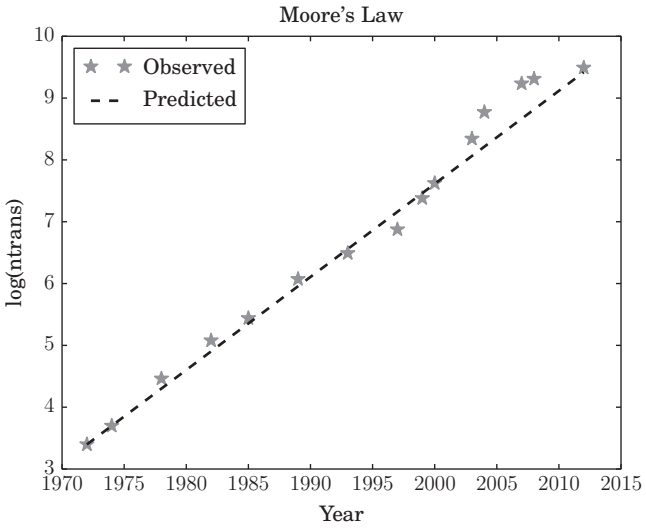

Figure 3.9 Moore's Law.

3.2.3 Exercises

Problems

P3.2.1 A molecule, A, reacts to form either B or C with first-order rate constants k_1 and k_2 respectively. That is,

$$\frac{\mathrm{d[A]}}{\mathrm{d}t} = -(k_1 + k_2)[\mathrm{A}],$$

and so

$$[\mathrm{A}] = [\mathrm{A}]_0 \mathrm{e}^{-(k_1+k_2)t},$$

where $[\mathrm{A}]_0$ is the initial concentration of A. The product concentrations (starting from 0), increase in the ratio $[\mathrm{B}]/[\mathrm{C}] = k_1/k_2$ and conservation of matter requires $[\mathrm{B}] + [\mathrm{C}] = [\mathrm{A}]_0 - [\mathrm{A}]$. Therefore,

$$[\mathrm{B}] = \frac{k_1}{k_1 + k_2}[\mathrm{A}]_0 \left(1 - \mathrm{e}^{-(k_1+k_2)t}\right)$$

$$[\mathrm{C}] = \frac{k_2}{k_1 + k_2}[\mathrm{A}]_0 \left(1 - \mathrm{e}^{-(k_1+k_2)t}\right)$$

For a reaction with $k_1 = 300\ \mathrm{s}^{-1}$ and $k_2 = 100\ \mathrm{s}^{-1}$, plot the concentrations of A, B and C against time given an initial concentration of reactant, $[\mathrm{A}]_0 = 2.0\ \mathrm{mol\,dm}^{-3}$.

P3.2.2 A *Gaussian integer* is a complex number whose real and imaginary parts are both integers. A *Gaussian prime* is a Gaussian integer $x + iy$ such that either:

- one of x and y is zero and the other is a prime number of the form $4n + 3$ or $-(4n + 3)$ for some integer $n \geq 0$; or
- both x and y are nonzero and $x^2 + y^2$ is prime.

Consider the sequence of Gaussian integers traced out by an imaginary particle, initially at c_0, moving in the complex plane according to the following rule: it takes integer steps in its current direction (± 1 in either the real or imaginary direction), but turns *left* if it encounters a Gaussian prime. Its initial direction is in the positive real direction ($\Delta c = 1 + 0i \Rightarrow \Delta x = 1, \Delta y = 0$). The path traced out by the particle is called a *Gaussian prime spiral*.

Write a program to plot the Gaussian prime spiral starting at $c_0 = 5 + 23i$.

P3.2.3 The annual risk of death (given as "1 in N") for men and women in the UK in 2005 for different age ranges is given in the table below. Use `pylab` to plot these data on a single chart.

Age range	Female	Male
< 1	227	177
1–4	5376	4386
5–14	10417	8333
15–24	4132	1908
25–34	2488	1215
35–44	1106	663
45–54	421	279
55–64	178	112
65–74	65	42
75–84	21	15
> 84	7	6

3.3 More advanced plotting

3.3.1 Polar plots

`pylab.plot` produces a plot on Cartesian (x, y) axes. To produce a polar plot using (r, θ) coordinates, use `pylab.polar`, which is passed arguments `theta` (which is usually the independent variable) and `r`.

Example E3.5 A cardioid is the plane figure described in polar coordinates by $r = 2a(1 + \cos\theta)$ for $0 \leq \theta \leq 2\pi$:

```
theta = pylab.linspace(0, 2.*pylab.pi, 1000)
a = 1.
r = 2 * a * (1. + pylab.cos(theta))
pylab.polar(theta, r)
pylab.show()
```

The polar graph plotted by this code is illustrated in Figure 3.10.

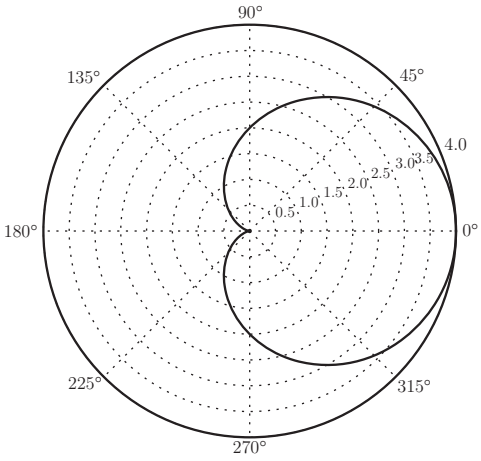

Figure 3.10 The cardioid figure formed with $a = 1$.

3.3.2 Histograms

A *histogram* represents the distribution of data as a series of (usually vertical) bars with lengths in proportion to the number of data items falling into predefined ranges (known as *bins*). That is, the range of data values is divided into intervals and the histogram constructed by counting the number of data values in each interval.

The `pylab` function `hist` produces a histogram from a sequence of data values. The number of bins can be passed as an optional argument, `bins`; its default value is 10. Also by default the height of the histogram bars are absolute counts of the data in the corresponding bin; setting the attribute `normed=True` normalizes the histogram so that its area (the height times width of each bar summed over the total number of bars) is unity.

For example, take 5,000 random values from the normal distribution with mean 0 and standard deviation 2 (see Section 4.5.1):

```
>>> import pylab
>>> import random
>>> data = []
>>> for i in range(5000):
...     data.append(random.normalvariate(0, 2))
>>> pylab.hist(data, bins=20, normed=True)
>>> pylab.show()
```

The resulting historgram is plotted in Figure 3.11.

3.3.3 Multiple axes

The command `pylab.twinx()` starts a new set of axes with the same x-axis as the original one, but a new y-scale. This is useful for plotting two or more data series, which share an abcissa (x-axis) but with y values which differ widely in magnitude or which have different units. This is illustrated in the following example.

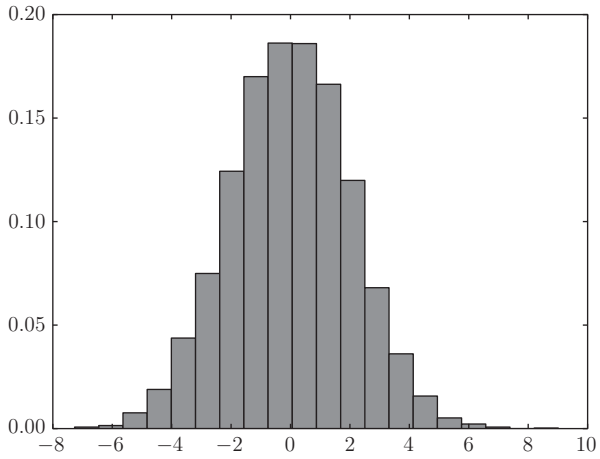

Figure 3.11 A histogram of random, normally distributed data.

Example E3.6 As described at http://tylervigen.com/, there is a curious but utterly meaningless correlation over time between the divorce rate in the US state of Maine and the per capita consumption of margarine in that country. The two time series here have different units and meanings and so should be plotted on separate *y*-axes, sharing a common *x*-axis (year).

Listing 3.3 The correlation between margarine consumption in the United States and the divorce rate in Maine

```
# eg3-margarine-divorce.py
import pylab

years = range(2000, 2010)
divorce_rate = [5.0, 4.7, 4.6, 4.4, 4.3, 4.1, 4.2, 4.2, 4.2, 4.1]
margarine_consumption = [8.2, 7, 6.5, 5.3, 5.2, 4, 4.6, 4.5, 4.2, 3.7]

❶ line1 = pylab.plot(years, divorce_rate, 'b-o',
                     label='Divorce rate in Maine')
pylab.ylabel('Divorces per 1000 people')
pylab.legend()

pylab.twinx()
line2 = pylab.plot(years, margarine_consumption, 'r-o',
                   label='Margarine consumption')
pylab.ylabel('lb of Margarine (per capita)')

# Jump through some hoops to get the both line's labels in the same legend:
❷ lines = line1 + line2
labels = []
for line in lines:
❸     labels.append(line.get_label())

pylab.legend(lines, labels)
pylab.show()
```

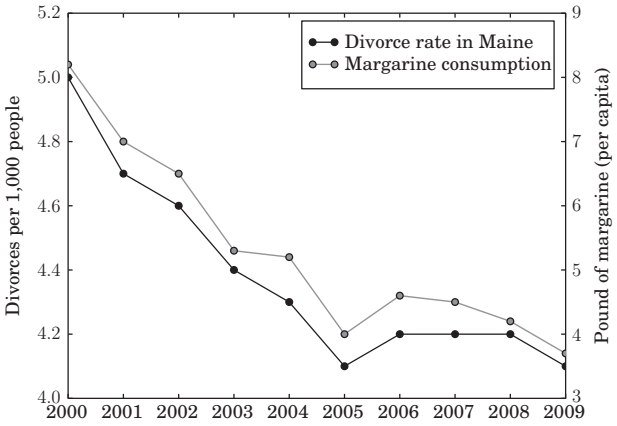

Figure 3.12 The correlation between the divorce rate in Maine and the per capita margarine consumption in the United States.

We have a bit of extra work to do in order to place a legend labeled with both lines on the plot: ❶ `pylab.plot` returns a list of objects representing the lines that are plotted, so we save them as `line1` and `line2`, ❷ concatenate them, and then ❸ loop over them to retrieve their labels. The list of lines and labels can then be passed to `pylab.legend` directly. The result of this code is the graph plotted in Figure 3.12.

3.3.4 Exercises

Problems

P3.3.1 A spiral may be considered to be the figure described by the motion of a point on an imaginary line as that line pivots around an origin at constant angular velocity. If the point is fixed on the line, then the figure described is a circle.

a. If the point on the rotating line moves from the origin with constant speed, its position describes an *Archimedean spiral*. In polar coordinates the equation of this spiral is $r = a + b\theta$. Use `pylab` to plot the spiral defined by $a = 0, b = 2$ for $0 \leq \theta \leq 8\pi$.

b. If the point moves along the rotating line with a velocity that increases in proportion to its distance from the origin, the result is a *logarithmic spiral*, which may be written as $r = a^{\theta}$. Plot the logarithmic spiral defined by $a = 0.8$ for $0 \leq \theta \leq 8\pi$. The logarithmic spiral has the property of *self-similarity*: with each 2π whorl, the spiral grows but maintains its shape.[3] Logarithmic spirals occur

[3] The Swiss mathematician Jakob Bernoulli was so taken with this property that he coined the logarithmic spiral *Spira mirabilis*: the "miraculus sprial" and wanted one engraved on his headstone with the phrase "Eadem mutata resurgo" ("Although changed, I shall arise the same"). Unfortunately, an Archimedian spiral was engraved by mistake.

frequently in nature, from the arrangements of the chambers of nautilus shells to the shapes of galaxies.

P3.3.2 A simple model for the interaction potential between two atoms as a function of their distance, r, is that of Lennard-Jones:

$$U(r) = \frac{B}{r^{12}} - \frac{A}{r^6},$$

where A and B are positive constants.[4]

For Argon atoms, these constants may be taken to be $A = 1.024 \times 10^{-23}$ J nm^6 and $B = 1.582 \times 10^{-26}$ J nm^{12}.

a. Plot $U(r)$. On a second y-axis on the same figure, plot the interatomic force

$$F(r) = -\frac{\mathrm{d}U}{\mathrm{d}r} = \frac{12B}{r^{13}} - \frac{6A}{r^7}.$$

Your plot should show the "interesting" part of these curves, which tend rapidly to very large values at small r.

Hint: life is easier if you divide A and B by Boltzmann's constant, 1.381×10^{-23} J K^{-1} so as to measure $U(r)$ in units of K. What is the depth, ϵ, and location, r_0, of the potential minimum for this system?

b. For small displacements from the equilibrium interatomic separation (where $F = 0$), the potential may be approximated to the harmonic oscillator function, $V(r) = \frac{1}{2}k(r - r_0)^2 + \epsilon$, where

$$k = \left|\frac{\mathrm{d}^2U}{\mathrm{d}r^2}\right|_{r_0} = \frac{156B}{r_0^{14}} - \frac{42A}{r_0^8}.$$

Plot $U(r)$ and $V(r)$ on the same diagram.

P3.3.3 The seedhead of a sunflower may be modeled as follows. Number the n seeds $s = 1, 2, \cdots, n$ and place each seed a distance $r = \sqrt{s}$ from the origin, rotated $\theta = 2\pi s/\phi$ from the x axis, where ϕ is some constant. The choice nature makes for ϕ is the *golden ratio*, $\phi = (1 + \sqrt{5})/2$, which maximizes the packing efficiency of the seeds as the seedhead grows.

Write a Python program to plot a model sunflower seedhead. (*Hint:* use polar coordinates.)

[4] This was popular in the early days of computing because r^{-12} is easy to compute as the square of r^{-6}.