Chapter 3

Graphics and visualization

So far we have created programs that print out words and numbers, but often we will also want our programs to produce graphics, meaning pictures of some sort. In this chapter we will see how to produce the two main types of computer graphics used in physics. First, we look at that most common of scientific visualizations, the graph: a depiction of numerical data displayed on calibrated axes. And second, we will see how to make scientific diagrams and animations: depictions of the arrangement or motion of the parts of a physical system, which can be useful in understanding the structure or behavior of the system.

3.1 Graphs

A number of Python packages include features for making graphs. In this book we will use the powerful and popular package `matplotlib`,[1] and particularly the submodule within `matplotlib` called `pyplot`, which creates standard two-dimensional plots. This module can make plots of a variety of different types. We will concentrate on three that are especially useful in physics: ordinary line graphs, scatter plots, and density (or heat) plots.[2]

There are a large number of functions in the `pyplot` module and, rather than import them individually when we need them, this is a situation where it makes sense to import them all at once with a statement of the form

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

(See Section 2.2.5 for a discussion of the `import` statement.) This gives us access to any function in the `pyplot` module. For instance, to use the basic `plot` function for

[1] Note that the name of the package is `matplotlib`—"`mat`" not "`math`". It is a common programming error to mistype the name of the package.

[2] One can also make contour plots, polar plots, pie charts, histograms, and more, and all of these find occasional use in physics. If you find yourself needing one of these more specialized graph types, you can find instructions for making them in the online documentation at `matplotlib.org`.

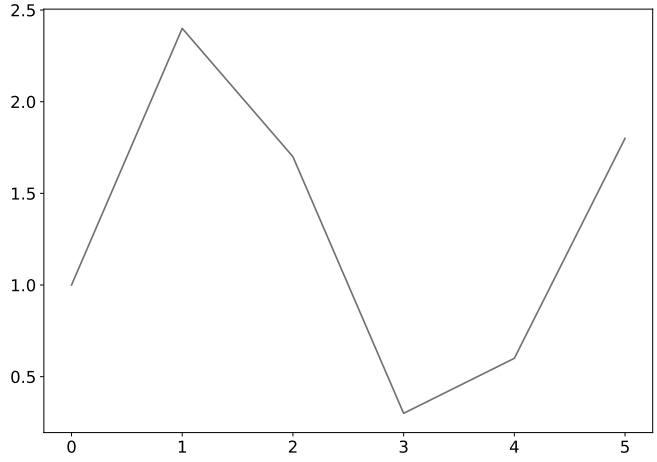
making a graph we would say `plt.plot`. Let us see how this function is used.

To create an ordinary graph we use the function `plot`. In the simplest case, this function takes one argument, which is a list or array of the values we want to plot. The function creates a graph of the given values in the memory of the computer, but it doesn't actually display it on the screen of the computer—it is stored in the memory but not yet visible to the computer user. To display the graph we use a second function from `pyplot`, the `show` function, which takes the graph in memory and draws it on the screen. Here is a complete program for plotting a small graph:

```
import matplotlib.pyplot as plt
y = [1.0, 2.4, 1.7, 0.3, 0.6, 1.8]
plt.plot(y)
plt.show()
```

After importing `pyplot` as `plt`, we create the list of values to be plotted, create a graph of those values with `plt.plot(y)`, then display that graph on the screen with `plt.show()`. Note that `plt.show()` has parentheses after it—it is a function that has no argument, but the parentheses still need to be there.

If we run the program above, it produces a new window on the screen with a graph in it like this:



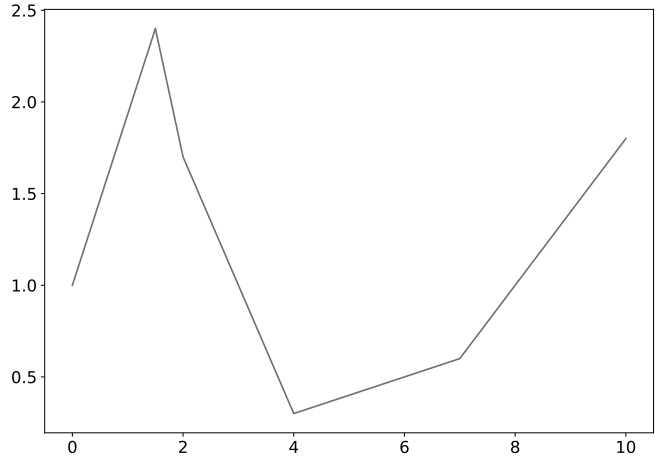
The computer has plotted the values in the list `y` at unit intervals along the x-axis (starting from zero in the standard Python style) and it has joined them up with straight lines.

While it's better than nothing, this is not a very useful kind of graph for physics purposes. Normally we want to specify both the x- and y-coordinates of the points in the graph. We can do this using a `plot` statement with two arguments, thus:

```
import matplotlib.pyplot as plt
x = [0.0, 1.5, 2.0, 4.0, 7.0, 10.0]
```

```
y = [1.0, 2.4, 1.7, 0.3, 0.6, 1.8]
plt.plot(x,y)
plt.show()
```

which produces a graph like this:



The first of the two arguments is a list specifying the x-coordinates of each of the points; the second specifies the y-coordinates. The computer plots the points at the given positions and then again joins them with straight lines. The two lists must have the same number of entries. If they do not, you will get an error message and no graph.

Why do we need two functions, `plot` and `show`, to make a graph? In the examples above it seems like it would be fine to combine the two into a single function that both creates a graph and shows it on the screen. However, there are more complicated situations where it is useful to have separate functions. In particular, in cases where we want to plot two or more different curves on the same graph, we can do so by using the `plot` function two or more times, once for each curve. Then we use the `show` function once to make a single graph with all the curves on it. We will see examples of this shortly.

Once you have displayed a graph on the screen you can do other things with it. The graph will appear in a window like the one shown in Fig. 3.1, with a number of buttons along the bottom. Among other things, these buttons allow you to zoom in on portions of the graph, move your view around the graph, or save the graph in various file formats, allowing you to view it again later, print it, or insert it as a figure in a document.

Now let us apply the `plot` and `show` functions to the creation of a slightly more interesting graph, a graph of the function $\sin x$ from $x = 0$ to $x = 10$. To do this we first create an array of the x values, then we take the sines of those values to get the y-coordinates of the points. Here is the program:

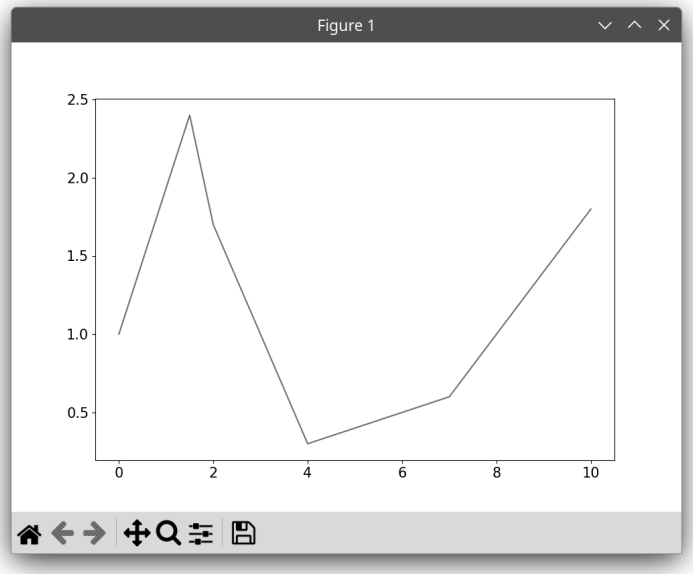


Figure 3.1: A graph window as it appears on the computer screen.

```
import matplotlib.pyplot as plt
from numpy import linspace,sin

x = linspace(0,10,100)
y = sin(x)
plt.plot(x,y)
plt.show()
```

Notice how we used the `linspace` function from `numpy` (see Section 2.5) to generate the array of x values, and the `sin` function from `numpy`, which is a special version that works with arrays—it takes the sine of every element in the array. (We could alternatively have used the ordinary `sin` function from the `math` package and taken the sines of each element individually using a for loop. As is often the case, there is more than one way to do the job.)

If we run this program we get the classic sine curve graph shown in Fig. 3.2. Note that we have not really drawn a curve at all here: our plot consists of a finite set of points—a hundred of them in this case—and the computer draws straight lines joining these points. So the end result is not actually curved; it is a set of straight-line segments. To us, however, it looks like a convincing sine wave because our eyes are not sharp enough to see the slight kinks where the segments meet. This is a useful

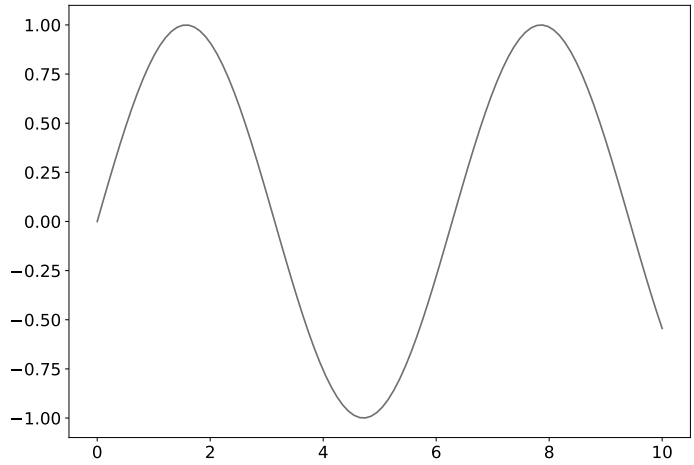


Figure 3.2: Graph of the sine function. A simple graph of the sine function produced by the program given in the text.

and widely used trick for making curves in computer graphics: choose a set of points spaced closely enough together that when joined with straight lines the result looks like a curve even though it really isn't.

As another example of the use of the `plot` function, suppose we have some experimental data in a computer file `values.txt`, stored in two columns, like this:

```
0        12121.71
1        12136.44
2        12226.73
3        12221.93
4        12194.13
5        12283.85
6        12331.60
7        12309.25
...
```

We can make a graph of these data as follows:

```
from numpy import loadtxt
import matplotlib.pyplot as plt

data = loadtxt("values.txt",float)
x = data[:,0]
y = data[:,1]
plt.plot(x,y)
plt.show()
```

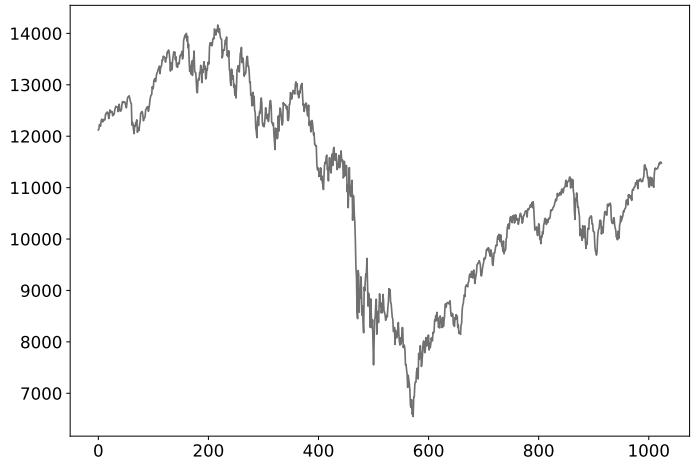


Figure 3.3: Graph of data from a file. This graph was produced by reading two columns of data from a file using the program given in the text.

Here we have used the `loadtxt` function from `numpy` (see Section 2.4.3) to read the values in the file and put them in an array and then we have used Python's array slicing facilities (Section 2.4.5) to extract the first and second columns of the array and put them in separate arrays `x` and `y` for plotting. The end result is a plot as shown in Fig. 3.3.

In fact, it is not necessary in this case to use the separate arrays `x` and `y`. We could shorten the program by saying instead

```
data = loadtxt("values.txt",float)
plt.plot(data[:,0],data[:,1])
plt.show()
```

which achieves the same result. Arguably, however, this is more difficult to read. As we emphasized in Section 2.7, readability is a defining quality of well-written programs, so you might in this case want to use the extra arrays `x` and `y` even though they are not strictly necessary.

An important point to notice about all of these examples is that the program *stops* when it displays the graph. To be precise it stops when it gets to the `show` function. Once you use `show` to display a graph, the program will go no further until you close the window containing the graph. Only once you close the window will the computer proceed with the next line of your program. The function `show` is said to be a *blocking* function—it blocks the progress of the program until the function is done with its job. We have seen one other example of a blocking function previously, the function `input`, which collects input from the user at the keyboard. It too halts the running of the program until its job is done. (The blocking action of the `show`

function has little impact in the programs above, since the show statement is the last line of the program in each case. But in more complex examples there might be further lines after the show statement and their execution would be delayed until the graph window was closed.)

A useful trick that we will employ frequently in this book is to build the lists of x- and y-coordinates for a graph step by step as we go through a calculation. It will happen often that we do not know all of the x or y values for a graph ahead of time. We work them out one by one as part of some calculation we are doing. In this case, a good way to proceed is to start with two empty lists for the x- and y-coordinates and add points to them one by one, as we calculate the values. Going back to the sine wave example, for instance, here is an alternative way to make a graph of $\sin x$ that calculates the individual values one by one and adds them to a growing list:

```
from math import sin
from numpy import linspace
import matplotlib.pyplot as plt

xpoints = []
ypoints = []
for x in linspace(0,10,100):
    xpoints.append(x)
    ypoints.append(sin(x))

plt.plot(xpoints,ypoints)
plt.show()
```

If you run it, you will find that this program produces a picture of a sine wave identical to the one in Fig. 3.2. Notice how we created the two empty lists and then appended values to the end of each of them, one by one, using a for loop. We will use this technique often. (See Section 2.4.1 for a discussion of the append function.)

The graphs we have seen so far are very simple, but there are many extra features we can add to them, some of which are illustrated in Fig. 3.4. For instance, in the previous graphs the computer chose the range of x and y values for the two axes. Normally the computer makes good choices, but occasionally you might like to make different ones. In our picture of a sine wave, Fig. 3.2, for instance, you might decide that the graph would be clearer if there were a little more space at the top and bottom of the curve. You can override the computer's choice of x- and y-axis limits with the functions `xlim` and `ylim`. These functions take two arguments each, for the lower and upper limits of the range of the respective axes. Thus, for instance, we might modify our sine wave program as follows:

```
import matplotlib.pyplot as plt
from numpy import linspace,sin
```

```
x = linspace(0,10,100)
y = sin(x)
plt.plot(x,y)
plt.ylim(-1.2,1.2)
plt.show()
```

The resulting graph is shown in Fig. 3.4a and, as we can see, it now has a little extra space above and below the curve because the y-axis has been modified to run from -1.2 to $+1.2$. Note that the `ylim` statement has to come after the `plot` statement but before the `show` statement—the `plot` statement has to create the graph first before you can modify its axes.

It is good practice to label the axes of your graphs, so that you and anyone else knows what they represent. You can add labels to the x- and y-axes with the functions `xlabel` and `ylabel`, which take a string argument—a string of letters or numbers in quotation marks. Thus we could modify our program as follows:

```
plt.plot(x,y)
plt.ylim(-1.2,1.2)
plt.xlabel("x axis")
plt.ylabel("y = sin x")
plt.show()
```

which produces the graph shown in Fig. 3.4b.

You can also vary the style in which the computer draws the curve on the graph. To do this a third argument is added to the `plot` function, which takes the form of a (slightly cryptic) string of characters, like this:

```
plt.plot(x,y,"ko-")
```

The first letter of the string tells the computer with what color to draw the curve. Allowed letters are `r`, `g`, `b`, `c`, `m`, `y`, `k`, and `w`, for red, green, blue, cyan, magenta, yellow, black, and white. The second letter specifies a symbol or marker to be placed at each data point along the curve. Possible symbols include `s`, `o`, `x`, `*`, and `.` for squares, circles, crosses, stars, and dots, respectively. The set of allowed symbols is quite large—see the online documentation at `matplotlib.org` for a full list. The third part of the string specifies the style of the line. Allowed styles include "`-`" for a solid line, "`--`" for a dashed line, and "`:`" for a dotted line. Any part of the string can be omitted. If the color is omitted the plot will use the default color, which is blue. If the symbol specifier is omitted, there will be no symbols at all. If the line style is omitted there will be no line. And if both symbol and line style are omitted then a solid line is used with no symbols.

Thus in the example above, the string `"ko-"` produces a graph in black with circular data points connected by solid lines. The result is shown in Fig. 3.4c. Conversely, `"rs--"` would produce a graph in red with squares connected by dashed

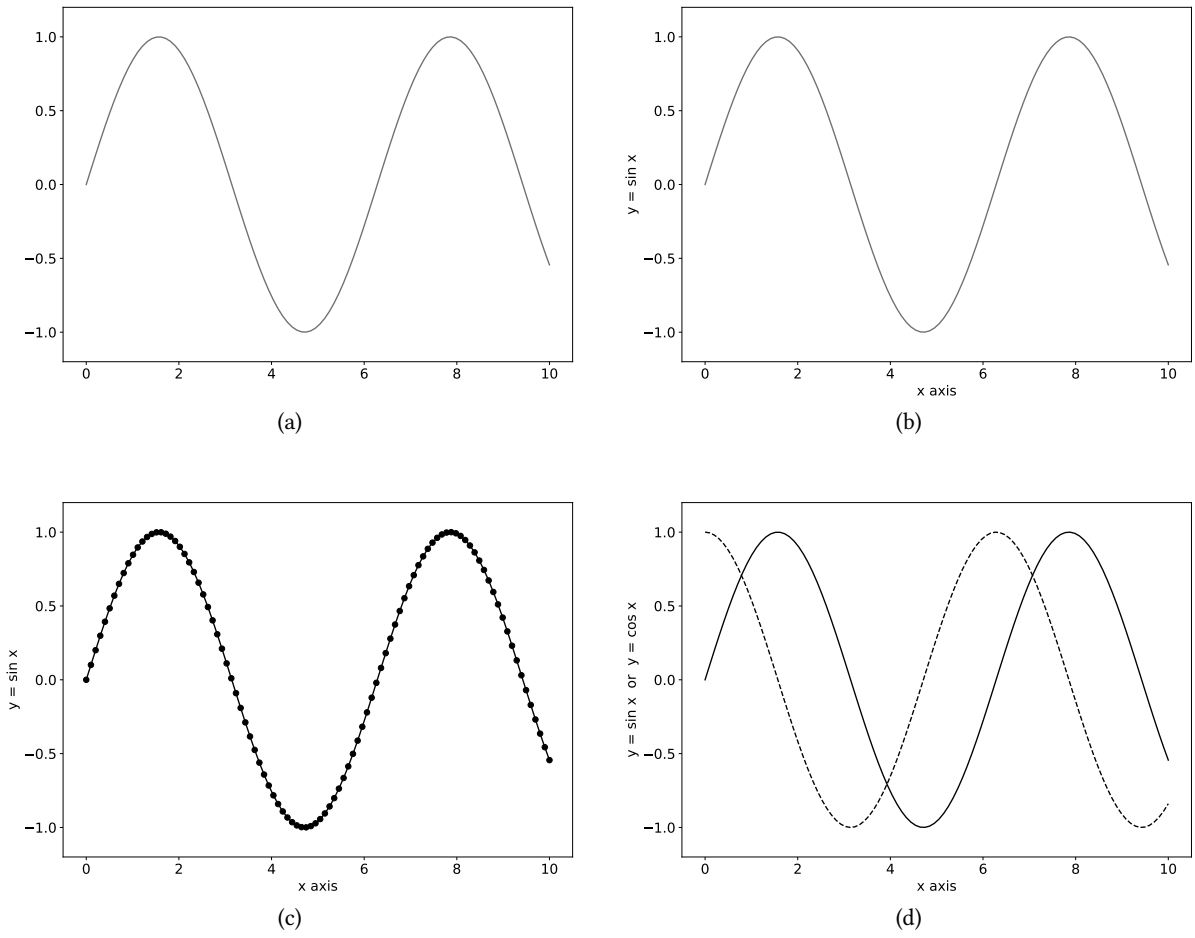


Figure 3.4: Graph styles. Several different versions of the same sine wave plot. (a) A basic graph, but with a little extra space added above and below the curve to make it clearer. (b) A graph with labeled axes. (c) A graph with the data points marked by circular dots. (d) Sine and cosine curves on the same graph.

lines, `"g*"` would produce green stars with no lines, and `"m"` on its own would produce a solid magenta line with no symbols.

Finally, we will often need to plot more than one curve or set of points on the same graph. This can be achieved by using the `plot` function repeatedly. For instance, here is a complete program that plots both the sine function and the cosine function on the same graph, one as a solid curve, the other as a dashed curve:

```
import matplotlib.pyplot as plt
from numpy import linspace,sin,cos
```

```
x = linspace(0,10,100)
y1 = sin(x)
y2 = cos(x)
plt.plot(x,y1,"k-")
plt.plot(x,y2,"k--")
plt.ylim(-1.2,1.2)
plt.xlabel("x axis")
plt.ylabel("y = sin x  or  y = cos x")
plt.show()
```

The result is shown in Fig. 3.4d.

This last example emphasizes the benefit of importing the entire `pyplot` module under the name `plt`. As we use more and more functions from the module to tune how our plot appears, it would become tedious to import each one individually. By importing the entire module at once we avoid this.

There are many other variations and styles available in `pyplot`. You can add legends and annotations to your graphs. You can change the color, size, or typeface used in the labels. You can change the color or style of the axes, or add a background color to the graph. These and many other possibilities are described in the online documentation at `matplotlib.org`.

Exercise 3.1: Plotting experimental data

In the online resources you will find a file called `sunspots.txt`, which contains the observed number of sunspots on the Sun for each month since January 1749. The file contains two columns of numbers, the first being the month and the second being the sunspot number.

a) Write a program that reads the data and makes a graph of sunspots as a function of time.

b) Modify your program to display only the first 1000 data points on the graph.

c) Modify your program further to calculate and plot the running average of the data, defined by

$$Y_k = \frac{1}{2r+1} \sum_{m=-r}^{r} y_{k+m}$$

where $r = 5$ in this case (and the y_k are the sunspot numbers). Have the program plot both the original data and the running average on the same graph, again over the range covered by the first 1000 data points.

Exercise 3.2: Curve plotting

Although the `plot` function is designed primarily for plotting standard xy graphs, it can be adapted for other kinds of plotting as well.

a) Make a plot of the so-called *deltoid* curve, which is defined parametrically by the equations

$$x = 2\cos\theta + \cos 2\theta, \qquad y = 2\sin\theta - \sin 2\theta,$$

where $0 \le \theta < 2\pi$. Take a set of values of θ between zero and 2π and calculate x and y for each from the equations above, then plot y as a function of x.

b) Taking this approach a step further, one can make a polar plot $r = f(\theta)$ for some function f by calculating r for a range of values of θ and then converting r and θ to Cartesian coordinates using the standard equations $x = r\cos\theta$, $y = r\sin\theta$. Use this method to make a plot of the Galilean spiral $r = \theta^2$ for $0 \le \theta \le 10\pi$.

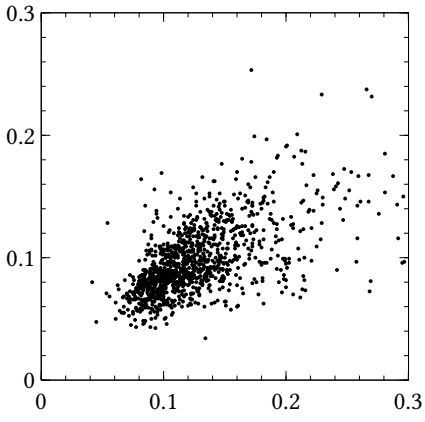
c) Using the same method, make a polar plot of "Fey's function"

$$r = e^{\cos\theta} - 2\cos 4\theta + \sin^5\frac{\theta}{12}$$

in the range $0 \le \theta \le 24\pi$.

3.2 SCATTER PLOTS

On an ordinary graph, such as those of the previous section, there is one independent variable, usually placed on the horizontal axis, and one dependent variable, on the vertical axis. The graph is a visual representation of the variation of the dependent variable as a function of the independent one—voltage as a function of time, say, or temperature as a function of position. In other cases, however, we measure or calculate two dependent variables. A classic example in physics is the temperature and brightness—also called the magnitude—of stars. Typically we might measure temperature and magnitude for each star in a given set and we would like some way to visualize how the two quantities are related. A standard approach is to use a *scatter plot*, a graph in which the two quantities are placed along the axes and we make a dot on the plot for each pair of measurements, i.e., for each star in this case.



A small scatter plot.

There are two different ways to make a scatter plot using the `pyplot` module. One of them we have already seen: we can make an ordinary graph, but with dots rather than lines to represent the data points. Assuming we have imported `pyplot` under the name `plt` as before, we could use a statement of the form

```
plt.plot(x,y,"ko")
```

to place a black circle at each point. A slight variant of the same idea is this:

```
plt.plot(x,y,"k.")
```

which will produce smaller dots.

Alternatively, `pyplot` provides the function `scatter`, which is designed specifically for making scatter plots. It works in a similar

fashion to the `plot` function: you give it two lists or arrays, one containing the x-coordinates of the points and the other containing the y-coordinates, and it creates the corresponding scatter plot. We just write

```
plt.scatter(x,y)
```

You do not have to give a third argument telling `scatter` to plot the data as dots—all scatter plots use dots automatically. As with the `plot` function, `scatter` only creates the scatter plot in the memory of the computer but does not display it on the screen. To display it you need to use the `show` function.

As an example, the file `stars.txt` in the online resources contains a list of the temperatures and magnitudes of a set of stars, like this:

```
4849.4 5.97
5337.8 5.54
4576.1 7.72
4792.4 7.18
5141.7 5.92
6202.5 4.13
...
```

The first column is the temperature and the second is the magnitude. Here is a Python program to make a scatter plot of these data:

File: `hrdiagram.py`

```
import matplotlib.pyplot as plt
from numpy import loadtxt

data = loadtxt("stars.txt",float)
x = data[:,0]
y = data[:,1]

plt.scatter(x,y)
plt.xlabel("Temperature")
plt.ylabel("Magnitude")
plt.xlim(0,13000)
plt.ylim(-5,20)
plt.show()
```

If we run this program it produces the plot shown in Fig. 3.5.

Many of the same variants illustrated in Fig. 3.4 for the `plot` function work for the `scatter` function also. In this program we used `xlabel` and `ylabel` to label the temperature and magnitude axes, and `xlim` and `ylim` to set the ranges of the axes. You can also change the size, style, and color of the dots and many other things. In addition, as with the `plot` function, you can use `scatter` two or more times in succession to plot two or more sets of data on the same graph, or you can use any

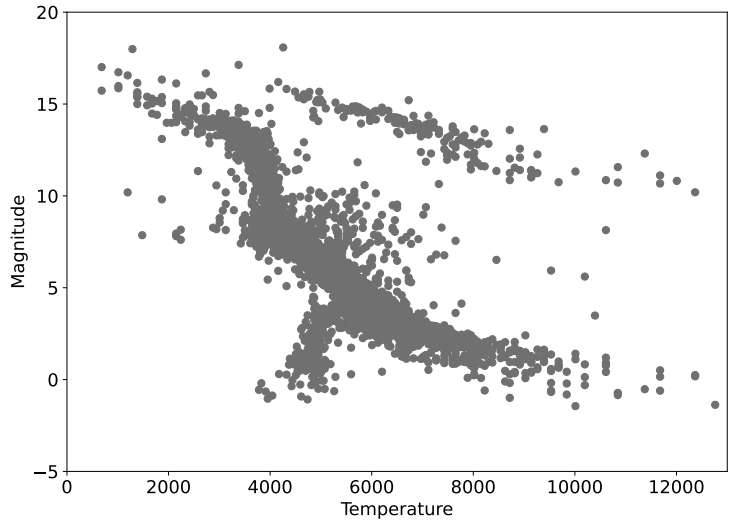


Figure 3.5: The Hertzsprung–Russell diagram. A scatter plot of the magnitude (i.e., brightness) of stars against their approximate surface temperature (which is estimated from the color of the light they emit). Each dot on the plot represents one star out of a catalog of 7860 stars that are close to our solar system.

combination of `scatter` and `plot` functions to draw scatter data and graphs on the same figure. Again, see the online manual at `matplotlib.org` for more details.

The scatter plot of the magnitudes and temperatures in Fig. 3.5 reveals an interesting pattern in the data: a substantial majority of the points lie along a rough band running from top left to bottom right of the plot. This is the so-called *main sequence* to which most stars belong. Rarer types of stars, such as red giants and white dwarfs, stand out in the figure as dots that lie off the main sequence. A scatter plot of stellar magnitude against temperature is called a *Hertzsprung–Russell diagram* after the astronomers who first drew it. The diagram is one of the fundamental tools of stellar astrophysics.

In fact, Fig. 3.5 is, in a sense, upside down, because the Hertzsprung–Russell diagram is, for historical reasons, normally plotted with both the magnitude and temperature axes *decreasing*, rather than increasing.[3] One of the nice things about `pyplot` is that it is easy to change this kind of thing with just a small modification

[3]The magnitude of a star is defined in such a way that it actually increases as the star gets fainter, so reversing the vertical axis makes sense since it puts the brightest stars at the top. The temperature axis is commonly plotted not directly in terms of temperature but in terms of the so-called *color index*, which is a measure of the color of light a star emits, which is in turn a measure of temperature. Temperature decreases with increasing color index, which is why the standard Hertzsprung–Russell diagram has temperature decreasing along the horizontal axis.

of the Python program. All we need to do in this case is change the `xlim` and `ylim` statements so that the start and end points of each axis are reversed, thus:

```
plt.xlim(13000,0)
plt.ylim(20,-5)
```

Then the figure will be magically turned around.

3.3 Density plots

There are many situations in physics when we are working with two-dimensional grids of data. A condensed matter physicist might measure variations in charge or temperature or atomic deposition on a solid surface, a fluid dynamicist might measure the heights of waves in a ripple tank, a particle physicist might measure the distribution of particles incident on an imaging detector, and so on. Two-dimensional data are harder to visualize on a computer screen than the one-dimensional lists of values that go into an ordinary graph. One tool that is helpful in many cases is the *density plot*, also sometimes called a *heat map*, a two-dimensional plot where color or brightness is used to indicate data values. Figure 3.6 shows an example.

In Python density plots are produced by the function `imshow` from `pyplot`. Here is the program that produced Fig. 3.6:

```
import matplotlib.pyplot as plt
from numpy import loadtxt

data = loadtxt("circular.txt",float)
plt.imshow(data)
plt.show()
```

The file `circular.txt` contains a simple array of values, like this:

```
0.0050 0.0233 0.0515 0.0795 0.1075 ...
0.0233 0.0516 0.0798 0.1078 0.1358 ...
0.0515 0.0798 0.1080 0.1360 0.1639 ...
0.0795 0.1078 0.1360 0.1640 0.1918 ...
0.1075 0.1358 0.1639 0.1918 0.2195 ...
  ...    ...    ...    ...    ...
```

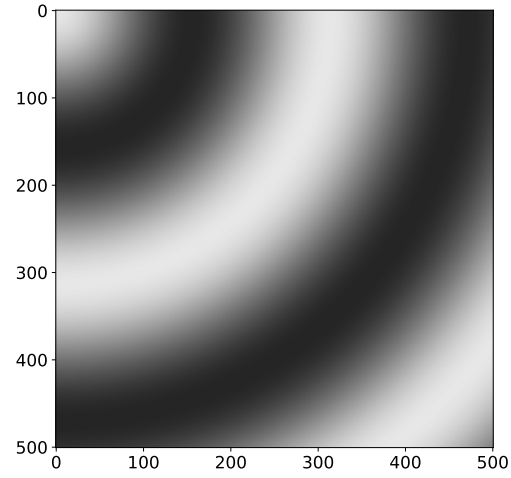


Figure 3.6: A example of a density plot

The program reads the values in the file and puts them in the two-dimensional array `data` using the `loadtxt` function, then creates the density plot with the `imshow` function and displays it with `show`. The computer automatically adjusts the color-scale so that the picture uses the full range of available shades.

The computer also adds numbered axes along the sides of the figure, which measure the rows and columns of the array, although it is possible to change the calibration of the axes to use different units. We will see how to do this in a moment. The image produced is a direct picture of the array, laid out in the usual fashion for matrices, row by row, starting at the top and working downwards. Thus the top left corner in Fig. 3.6 represents the value stored in the array element `data[0,0]`, followed to the right by `data[0,1]`, `data[0,2]`, and so on. Immediately below those, the next row is `data[1,0]`, `data[1,1]`, `data[1,2]`, and so on.

The numerical labels on the axes reflect the array indices, meaning that the origin of the figure is at the top left and the vertical axis increases downward. While this is natural from the point of view of matrices, it is a little odd for a graph. Most of us are accustomed to graphs whose vertical axes increase upwards. What's more, the array elements `data[i,j]` are written (as is the standard with matrices) with the *row* index first—i.e., the vertical index—and the column, or horizontal, index second. This is the opposite of the convention used with graphs where coordinates are normally in x, y order—i.e., horizontal first, then vertical. There is nothing much we can do about these conventions: they are the ones that mathematicians settled upon centuries ago and it is too late to change them now. But the conflict between them can be confusing, so take this opportunity to make a mental note.

In fact, Python provides a way to deal with the first problem, of the origin in a density plot being at the top. You can include an additional argument with the `imshow` function thus:

```
plt.imshow(data,origin="lower")
```

which flips the density plot top-to-bottom, putting the array element `data[0,0]` in the lower left corner, as is conventional, and changing the labeling of the vertical axis accordingly, so that it increases in the upward direction. The resulting plot is shown in Fig. 3.7a. We will use this trick for most of the density plots in this book. Note, however, that this does not fix our other problem: the indices `i` and `j` for the element `data[i,j]` still correspond to vertical and horizontal positions respectively, not the reverse. That is, the index `i` corresponds to the y-coordinate and the index `j` corresponds to the x-coordinate. You need to keep this in mind when making density plots—it is easy to get the axes swapped by mistake.

The black-and-white visuals in this book do not really do justice to the density plot in Fig. 3.7a. The original is in bright colors, ranging through the spectrum from dark blue for the lowest values to yellow for the highest. If you wish, you can run the program yourself to see the density plot in its full glory—both the program, which is called `circular.py`, and the data file `circular.txt` can be found in the online resources. This color scheme is the default choice for density maps in Python, but it is not always the best. In fact, for most purposes, a simple gray-scale from black to white is easier to read. Luckily, it is simple to change the color scheme. To change

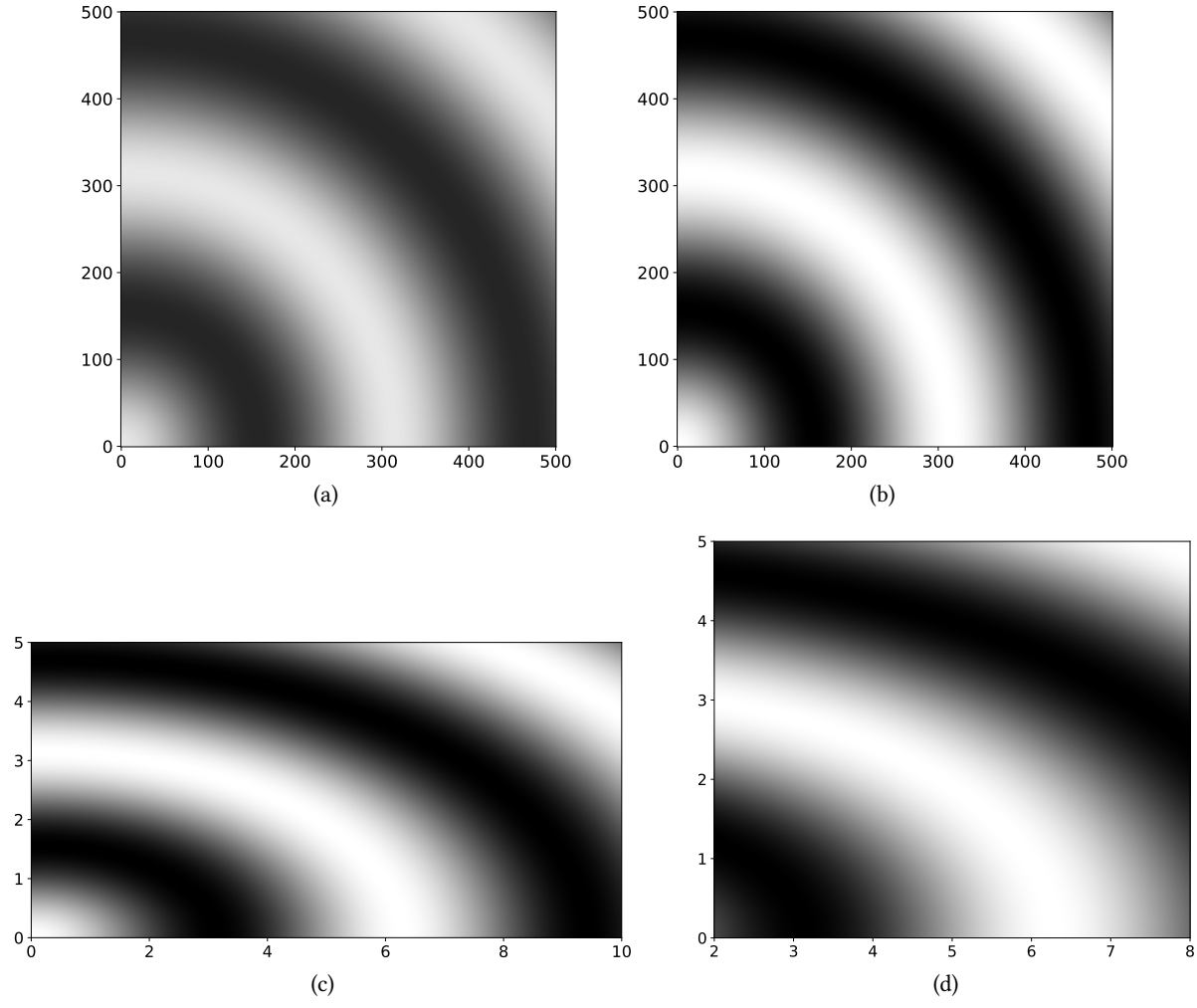


Figure 3.7: Density plots. Four different versions of the same density plot. (a) A plot with the vertical axis flipped so that the numbers increase up the page rather than down. (b) The same plot, but using the gray color scheme, which runs from black for the lowest values to white for the highest. (c) A plot with the calibration of the axes changed. Because the range chosen is different for the horizontal and vertical axes, the computer has altered the shape of the figure to keep distances equal in all directions. (d) The same plot as in (c) but with the horizontal range reduced so that only the middle portion of the data is shown.

to gray-scale, for instance, you use the function gray, which takes no arguments:[4]

```
import matplotlib.pyplot as plt
from numpy import loadtxt

data = loadtxt("circular.txt",float)
plt.imshow(data,origin="lower")
plt.gray()
plt.show()
```

Figure 3.7b shows the result. In black-and-white it looks pretty similar to the color version in panel (a), but on the screen it looks entirely different. Try it if you like.

All of the density plots in this book use the gray scale (except Figs. 3.6 and 3.7a). It may not be flashy, but it is informative, easy to read, and suitable for printing on black-and-white printers or for publications that are in black-and-white only (like this book and many scientific journals). However, pyplot provides many other color schemes, which you may find useful occasionally. A complete list, with illustrations, is given in the online documentation at matplotlib.org, but here are a few that might find use in physics:

viridis	The default blue and yellow color scheme
jet	A "heat map" color scheme that goes from blue to red
gray	Gray-scale running from black to white
hot	An alternative heat map that goes black-red-yellow-white
spectral	A spectrum with 7 clearly defined colors, plus black and white
bone	An alternative gray-scale with a hint of blue
hsv	A rainbow scheme that starts and ends with red

Each of these has a corresponding function, viridis(), jet(), and so forth, that selects the relevant color scheme for use in future plots. Many more color schemes are given in pyplot and one can also define one's own schemes, although the definitions involve some slightly tricky programming. Example code is given in Appendix C and in the online resources to define three additional schemes that can be useful for physics:[5]

redblue	Runs from red to blue via black
redwhiteblue	Runs from red to blue via white
inversegray	Runs from white to black, the opposite of gray

[4]The function gray works slightly differently from other functions we have seen that modify plots, such as xlabel or ylim. Those functions modified only the current plot, whereas gray (and the other color scheme functions in pyplot) changes the color scheme for all subsequent density plots. If you write a program that makes more than one plot, you only need to call gray once.

[5]To use these color schemes copy the file colormaps.py from the online resources into the folder containing your program and then in your program say, for example, "from colormaps import redblue". Then the statement "redblue()" will switch to the redblue color map.

There is also a function `colorbar()` in the `pyplot` module that instructs Python to add a bar to the side of your figure showing the range of colors used in the plot, along with a numerical scale indicating which values correspond to which colors, something that can be helpful when you want to make a more precise quantitative reading of a density plot.

As with graphs and scatter plots, you can modify the appearance of density plots in various ways. The functions `xlabel` and `ylabel` work as before, adding labels to the two axes. You can also change the scale marked on the axes. By default, the scale corresponds to the elements of the array holding the data, but you might want to calibrate your plot with a different scale. You can do this by adding an extra parameter to `imshow`, like this:

```
plt.imshow(data,origin="lower",extent=[0,10,0,5])
```

which results in a modified plot as shown in Fig. 3.7c. The argument consists of "`extent=`" followed by a list of four values, which give, in order, the beginning and end of the horizontal scale and the beginning and end of the vertical scale. The computer will use these numbers to mark the axes, but the actual content displayed in the body of the density plot remains unchanged—the `extent` argument affects only how the plot is labeled. This trick can be very useful if you want to calibrate your plot in "real" units. If the plot is a picture of the surface of the Earth, for instance, you might want axes marked in units of latitude and longitude; if it is a picture of a surface at the atomic scale you might want axes marked in nanometers.

Note also that in Fig. 3.7c the computer has changed the shape of the plot—its *aspect ratio*—to accommodate the fact that the horizontal and vertical axes have different ranges. The `imshow` function attempts to make unit distances equal along the horizontal and vertical directions where possible. Sometimes, however, this is not what we want, in which case we can tell the computer to use a different aspect ratio. For instance, if we wanted the present figure to remain square we would say:

```
plt.imshow(data,origin="lower",extent=[0,10,0,5],aspect=2.0)
```

This tells the computer to use unit distances twice as large along the vertical axis as along the horizontal one, which will make the plot square once more. You can also specify the special value `aspect="auto"`, which instructs the computer to choose the aspect ratio automatically to match the shape of the window on the screen. You can change the shape of the window after making the plot by dragging the window corners with your mouse, and the aspect ratio will change with it, which allows you to manually adjust the aspect ratio to get whatever result you want.

Note that we are free to use any or all of the `origin`, `extent`, and `aspect` arguments together in the same function. We don't have to use them all if we don't want to—any selection is allowed—and they can come in any order. We can also limit our density plot to just a portion of the data using the functions `xlim` and `ylim`, as with graphs and scatter plots. These functions work with the scales specified by the

`extent` argument, if there is one, or with the row and column indices otherwise. So, for instance, we could say `plt.xlim(2,8)` to reduce the density plot of Fig. 3.7b to just the middle portion of the horizontal scale, from 2 to 8. Figure 3.7d shows the result. Note that, unlike the `extent` argument, `xlim` and `ylim` do change which data are displayed in the body of the density plot—the `extent` argument makes purely cosmetic changes to the labeling of the axes, but `xlim` and `ylim` actually change which data appear.

Finally, you can use the functions `plot` and `scatter` to superimpose graphs or scatter plots of data on the same axes as a density plot. You can use any combination of `imshow`, `plot`, and `scatter` in sequence, followed by `show`, to create a single graph with density data, curves, or scatter data, all on the same set of axes.

Example 3.1: Wave interference

Suppose we drop a pebble into a pond and ripples radiate out from the spot where it fell. We could create a simple representation of the physics with a sine wave, spreading out in a uniform circle, to represent the height of the waves at some later time. If the center of the circle is at x_1, y_1 then the distance r_1 to the center from a point x, y is

$$r_1 = \sqrt{(x - x_1)^2 + (y - y_1)^2} \tag{3.1}$$

and the sine wave for the height is

$$\xi_1(x, y) = \xi_0 \sin kr_1 \,, \tag{3.2}$$

where ξ_0 is the amplitude and k is the wavevector, related to the wavelength λ by $k = 2\pi/\lambda$.

Now suppose we drop another pebble in the pond, creating another set of circular waves with the same wavelength and amplitude but centered on a different point x_2, y_2:

$$\xi_2(x, y) = \xi_0 \sin kr_2 \qquad \text{with} \qquad r_2 = \sqrt{(x - x_2)^2 + (y - y_2)^2}. \tag{3.3}$$

Then, assuming the waves add linearly (which is a reasonable assumption for water waves, provided they are not too big), the total height of the surface at point x, y is

$$\xi(x, y) = \xi_0 \sin kr_1 + \xi_0 \sin kr_2. \tag{3.4}$$

Suppose the wavelength of the waves is $\lambda = 5\,\text{cm}$, the amplitude is 1 cm, and the centers of the circles are 20 cm apart. Here is a program to make an image of the height over a 1 meter square region of the pond. To make the image we create an array of values representing the height ξ at a grid of points and then use that array to make a density plot. In this example we use a grid of 500×500 points to cover the 1 m square, which means the grid points have a separation of $100/500 = 0.2\,\text{cm}$.

File: ripples.py

```
from math import sqrt,sin,pi
from numpy import empty
import matplotlib.pyplot as plt

wavelength = 5.0
k = 2*pi/wavelength
xi0 = 1.0
separation = 20.0        # Separation of centers in cm
side = 100.0             # Side of the square in cm
points = 500             # Number of grid points along each side
spacing = side/points    # Spacing of points in cm

# Calculate the positions of the centers of the circles
x1 = side/2 + separation/2
y1 = side/2
x2 = side/2 - separation/2
y2 = side/2

# Make an array to store the heights
xi = empty([points,points],float)

# Calculate the values in the array
for i in range(points):
    y = spacing*i
    for j in range(points):
        x = spacing*j
        r1 = sqrt((x-x1)**2+(y-y1)**2)
        r2 = sqrt((x-x2)**2+(y-y2)**2)
        xi[i,j] = xi0*sin(k*r1) + xi0*sin(k*r2)

# Make the plot
plt.imshow(xi,origin="lower",extent=[0,side,0,side])
plt.gray()
plt.show()
```

This is the longest and most involved program we have seen so far, so it may be worth taking a moment to make sure you understand how it works. Note in particular how the height is calculated and stored in the array `xi`. The variables `i` and `j` go through the rows and columns of the array respectively, and from these we calculate the values of the coordinates x and y. Since, as discussed earlier, the rows correspond to the vertical axis and the columns to the horizontal axis, the value of x is calculated from `j` and the value of y is calculated from `i`. Other than this subtlety,

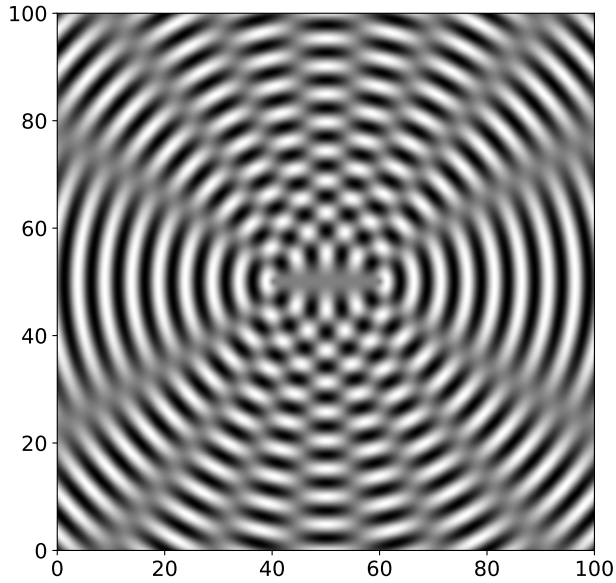


Figure 3.8: Interference pattern. This plot, produced by the program given in the text, shows the superposition of two circular sets of sine waves, creating an interference pattern with fringes that appear as the gray bars radiating out from the center of the picture.

the program is a fairly straightforward translation of Eqs. (3.1) to (3.4).[6]

If we run the program above, it produces the picture in Fig. 3.8, showing clearly the interference of the two sets of waves. Interference fringes are visible as the gray bands radiating from the center.

Exercise 3.3: There is a file in the online resources called `stm.txt`, which contains a grid of values from scanning tunneling microscope measurements of the (111) surface of silicon. A scanning tunneling microscope is a device that measures the shape of a surface at the atomic level by tracking a sharp tip over the surface and measuring quantum tunneling current as a function of position. The end result is a grid of values that represent the height of the surface and the file `stm.txt` contains just such a grid of values. Write a program that reads the data contained in the file and makes a density plot of the values. Use the various options and

[6] One other small detail is worth mentioning. We called the variable for the wavelength "`wavelength`". You might be tempted to call it "`lambda`" but if you did you would get an error message and the program would not run. The word lambda has a special meaning in the Python language and cannot be used as a variable name, just as words like "for" and "if" cannot be used as variable names. (See footnote 4 on page 15.) The names of other Greek letters—alpha, beta, gamma, and so on—are allowed as variable names.

variants you have learned about to make a picture that shows the structure of the silicon surface clearly.

3.4 Drawing

One of the flashiest applications of computers today is computer animation. In any given week millions of people flock to cinemas worldwide to watch the latest computer-animated movie from the big animation studios. Graphics and animation find a more humble, but very useful, application in computational physics as a tool for visualizing the behavior of physical systems.

There are a number of different packages available for making general drawings and animations in Python, some of them very complex. In this book we will use the package `qdraw`, a simpler package intended specifically for the sort of scientific diagrams and animations we will be making.[7] The package works by creating specified objects on the screen, such as circles, squares, lines, and so forth, and then optionally changing their position or orientation to make them move around. Here is a short first program using the package:

```
from qdraw import window,circle,show
window(xlim=[-1,1],ylim=[-1,1])
circle(pos=[0,0],size=1)
show()
```

When we run this program a window appears on the screen with a circle in it, as shown in Fig. 3.9. The program has done a number of things. First, the `window` function creates the window on your screen and also specifies the axes of the window. The `xlim` and `ylim` arguments give the beginning and end of the horizontal and vertical scales respectively, so in this case both axes run from −1 to +1. We use these coordinates to place and move objects in the window.

Next, the `circle` function creates a circle with a given position and size. The `pos` argument gives the position of the center of the circle—right at the origin in this case—and the `size` argument gives the diameter of the circle. The arguments of the `window` and `circle` functions in our example are all integers, but this is not required. They can also be floating-point numbers.

The `circle` function only creates the circle in the memory of the computer, in a manner similar to the `plot` function in `pyplot`. The circle does not actually appear until we call the `show` function in the last line of the program. This function serves a purpose similar to the `show` function in `pyplot`. It draws the circle on the screen and then "blocks" the program so that you can see the results: the program pauses at the `show` function until you close the graphics window, then continues, although

[7] The `qdraw` package was written by the author of this book, but employs Python's native "turtle" graphics engine, a standard part of the Python language.

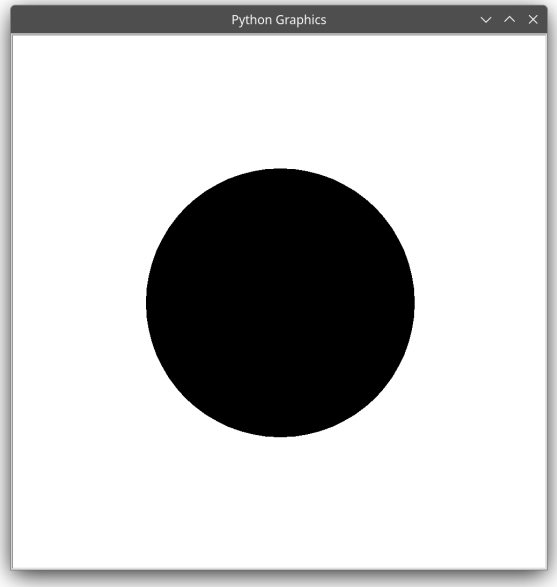


Figure 3.9: The circle drawn by the `qdraw` example in the text.

in most cases the `show` function will be the last line of the program anyway, so the program just finishes once the window is closed. In this simple program, it may seem pointless to have separate `circle` and `show` functions, but, like the corresponding construction in `pyplot`, the `show` function is useful for assembling more complex diagrams. It allows you to place many objects one after another and then visualize them all by calling the `show` function once.

Both the `window` and `circle` functions can have additional arguments that specify various details. For the `window` function we have:[8]

`width`	Specifies the width of the window in pixels
`height`	Specifies the height of the window in pixels
`bgcolor`	Specifies the background color of the window

For the `circle` function we have:

[8]It is possible to specify values for `width` and `height` that impose a different aspect ratio on the window than the one implied by `xlim` and `ylim`, effectively forcing the pixels in the window to be non-square. Python does not handle this situation well, producing distorted or glitchy images. To avoid this, you should specify at most three out of `width`, `height`, `xlim`, and `ylim`. If you specify less than three, the computer will use sensible defaults for the others. If you specify all four, it will ignore the `height` and behave as if only the other three were given. In most cases, giving `xlim` and `ylim` only will give good results.

`color`	Specifies the color of the circle
`olcolor`	Specifies the color of the outline of the circle
`olwidth`	Specifies the width of the outline in pixels

Documentation for these functions and others in the `qdraw` package is given in Appendix B on page 568.

Colors in `qdraw` can be given as strings surrounded by quotation marks and specified in various ways. You can use single letters as in `pyplot`: `r`, `g`, `b`, `c`, `m`, `y`, `k`, and `w` for red, green, blue, cyan, magenta, yellow, black, and white, respectively. You can also use complete words, like `"black"` or `"orange"`. Or, if you are an expert, you can use full RGB colors specified either as hexadecimal strings of the form `"#12ab34"` or as trios of real numbers `[0.22,0.75,0.31]` with each number between zero and one. Finally, colors can be specified as a single real number between zero and one that gets converted into a color using one of the `matplotlib` color schemes, such as the default blue-yellow "viridis" color scheme or the gray-scale color scheme—see Section 3.3. The color scheme and the range of numbers used in the mapping can be specified using the function `setcmap`—see Appendix B again.

The `qdraw` package also includes squares, rectangles, ellipses, and arbitrary polygons with any shape or size. Here are the functions that create each of these objects:

```
square(size=s,pos=[x,y])
rectangle(left=l,right=r,bottom=b,top=t,pos=[x,y])
ellipse(width=w,height=h,pos=[x,y])
polygon([[x1,y1],[x2,y2],...],pos=[x,y])
```

For a detailed explanation of the meaning of all the parameters and full documentation take a look at Appendix B. In addition to the parameters above, ones like `color` and `olcolor` can also be used to give the objects a different appearance.

By placing many objects on the screen in different positions and orientations, one can build up complex diagrams. Note that when we place multiple objects, the later ones are drawn on top of the earlier ones and may partially hide them from sight. When all the objects are in black, which is the default, this has no effect, since they all blend together anyway, but it can be noticeable if they have different colors.

Example 3.2: Picturing an atomic lattice

Suppose we have a solid composed of atoms arranged on a simple square lattice. We can visualize the arrangement of the atoms using the `qdraw` package by creating a picture with many circles at integer positions (i, j) with $i, j = -L \ldots L$, thus:

File: `lattice.py`

```
from qdraw import window,circle,show

L = 5
s = 0.3
```

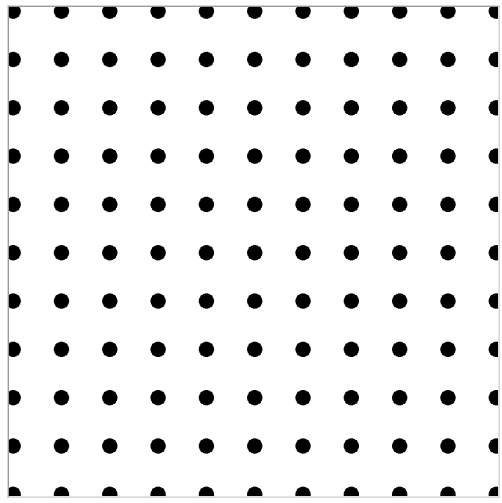


Figure 3.10: Visualization of atoms on a square lattice. A simple drawing of atoms on a square lattice, generated using the program in the text.

```
window(xlim=[-L,L],ylim=[-L,L])
for i in range(-L,L+1):
    for j in range(-L,L+1):
        circle(pos=[i,j],size=s)
show()
```

Observe how this program uses two nested for loops to run through all combinations of the values of `i` and `j`. When we run this program it produces the picture shown in Fig. 3.10. Download the program and try it if you like. You can also change the value of `L` to make a bigger lattice or `s` to alter the sizes of the atoms.

Of course real atomic lattices are (mostly) three-dimensional, whereas our simple picture here is two-dimensional. Nonetheless such visualizations can be helpful. See Exercise 3.4 below for some examples.

Exercise 3.4: Using the program from Example 3.2 above as a starting point, or starting from scratch if you prefer, create the following visualizations.

a) A sodium chloride crystal has sodium and chlorine atoms arranged on a square lattice, but the atoms alternate between sodium and chlorine in checkerboard fashion, so that each sodium is surrounded by chlorines and each chlorine is surrounded by sodiums. Create a visualization of (a two-dimensional) sodium chloride lattice using two different colors to represent the two types of atoms.

b) The face-centered cubic (fcc) lattice is the most common lattice in naturally occurring crystals. It consists of a square lattice with an additional atom at the center of each square. Create a two-dimensional visualization of an fcc lattice with a single species of atom (such as occurs in metallic nickel, for instance).

3.5 Animation

It is possible to change the properties of a circle or other shape after it is first created, including its position, size, and color. By doing so repeatedly, we can make the object appear to move on the screen and hence create an animation of a moving system. We will use this approach extensively in this book to visualize the workings of physical systems, such as the swing of a pendulum or the motions of the planets.

Consider a circle again. In order to change the properties of a circle on the screen after it is created, we use a slightly different form of the `circle` function, like this:

```
c = circle(size=1,pos=[0,0])
```

This form, in addition to placing a circle on the screen, creates a variable `c` in a manner similar to the way functions like `zeros` or `empty` create arrays (see Section 2.4.2). The new variable `c` is a variable of type "`circle`," in the same way that other variables are of type `int` or `float`.[9] This is a special variable type used only in the `qdraw` package to store the properties of circles and we can change those properties by performing operations on the variable. Thus, for example, we can say

```
from qdraw import window,circle,draw

window(xlim=[-1,1],ylim=[-1,1])
c = circle(size=1,pos=[0,0])
c.setpos(0.5,0.5)
draw()
```

The statement `c.setpos(0.5,0.5)` moves the center of the circle to new coordinates $(0.5, 0.5)$, but the change only occurs in the memory of the computer and not (yet) on the screen. That is what the `draw` statement is for: it tells the computer to draw all objects with their current positions. If the position of an object has changed, as here, this means the object will *move* on the screen. If we have multiple objects on the screen at once we can change all of their positions and then do `draw()` once and they will all move simultaneously.

The `draw` function is similar to the `show` function, but with a crucial difference: it is not a blocking function. It updates the drawing on the screen and then the program

[9] Technically, it is a Python `Turtle` object. See Appendix B and the documentation for the `turtle` package at `https://docs.python.org/3/library/turtle.html` if you want to learn more.

continues. This allows us to make a series of changes to the position of an object or objects, each followed by `draw()`, and create an animation.

To illustrate this approach let us make a simple animation of a circle which is itself looping around in a larger circle. The steps involved in doing this are: (1) create the circle, (2) go round a for loop and calculate the position of the circle at each step, then (3) update the position and call the `draw` function on each step to move the circle on the screen. Here is what the code looks like:

File: `revolve.py`

```
from qdraw import window,circle,draw
from math import cos,sin,pi
from numpy import arange

window(xlim=[-1.1,1.1],ylim=[-1.1,1.1])
c = circle(size=0.2,pos=[1,0])
for theta in arange(0,10*pi,0.02):
    x = cos(theta)
    y = sin(theta)
    c.setpos(x,y)
    draw()
```

Obviously we cannot illustrate this moving animation on the static page of this book. You will have to run the program yourself, but if you do you will discover that, while it does work, it is not very useful because the circle moves so fast you can barely see it. On any reasonably fast computer the program will make hundreds of moves per second and the moving circle will just be a blur. To overcome this issue, the `draw` function can pause for a moment each time it updates the screen. We can say

```
draw(0.01)
```

and the program will pause for 0.01 seconds each time around the loop, meaning that the animation will have a *framerate* of 100 frames per second, more than enough to ensure a smooth motion of the circle without being too fast. The version of the program in the online resources includes this modification and produces a good animation of a revolving circle. This simple program could be the basis, for instance, for an animation of the simultaneous motions of the planets of the solar system. Exercise 3.7 below invites you to create such an animation.

Another nice touch is to add the line

```
c.trail(length=30)
```

immediately after the circle is first created with the `circle` function. This statement turns on a "trail," which is a sort of short streamer that gets dragged behind the circle as it moves, so that you can see where the circle has been. Give it a try. The length parameter controls how long the streamer is: the length is specified as an integer n

and the streamer shows the last n movements of the object. If no length parameter is provided then the trail has no length limit—it records every movement of the object for as long as the animation lasts. When we look at the more complicated motions of real physical systems, this feature will be useful for clear visualization of those motions.

In addition to position there are various other properties of objects that we can change, such as color, rotation, and whether they are visible at all. For example, the statement `c.setcolor("green")` will change the color of our circle to green. And `c.visible(False)` will make the circle invisible, though not actually delete it—it still exists and maintains its position, and it can be made to appear again with `c.visible(True)`.

If we say `c.setangle(0.5)` the orientation of the object will be rotated to +0.5 radians from its initial position, in a counterclockwise direction. If our object is a circle this has no visible effect—a circle does not change when we rotate it. But for other objects it does make a difference. Here is a program, for instance, that produces an animation of a rotating square:

File: `spin.py`

```
from qdraw import window,square,draw
from math import pi
from numpy import arange

window(xlim=[-1.1,1.1],ylim=[-1.1,1.1])
s = square(size=1)
for theta in arange(0,20*pi,0.02):
    s.setangle(theta)
    draw(0.01)
```

There are a number of other features of the `qdraw` package that we will describe as the need arises later in the book, but for now this introduction will be enough to get us started.

Exercise 3.5: Lissajous figures

In the program `revolve.py` above we calculated the position of the circle as $x = \cos\theta$, $y = \sin\theta$. Modify the program so that instead it uses $x = \cos m\theta$, $y = \cos n\theta$, where m and n are positive integers. Also add a trail to the circle using `c.trail()`. Run the program with $m = 1$, $n = 2$ and with $m = 2$, $n = 3$ and observe the motion you get. This type of motion is called a *Lissajous figure*, after French physicist Jules Lissajous.

Exercise 3.6: Make a program in which there are N circles, equally spaced in a larger circle and all rotating around it at the same speed. N should be a variable and your program should be written so that you can change the value of N in only one place and the number of circles

around the loop will automatically change. Give some thought to how you are going to store the N circle objects, and what the formulas are for the x and y positions of each circle.

Exercise 3.7: Visualization of the solar system

The innermost six planets of our solar system revolve around the Sun in roughly circular orbits that all lie approximately in the same (ecliptic) plane. Here are some basic parameters:

Object	Radius of object (km)	Radius of orbit (millions of km)	Period of orbit (days)
Mercury	2440	57.9	88.0
Venus	6052	108.2	224.7
Earth	6371	149.6	365.3
Mars	3386	227.9	687.0
Jupiter	69173	778.5	4331.6
Saturn	57316	1433.4	10759.2
Sun	695500	–	–

Using the `qdraw` package, create an animation of the solar system that shows the following:

a) The Sun and planets as circles in their appropriate positions and with sizes proportional to their actual sizes. Because the radii of the planets are tiny compared to the distances between them, represent the planets by circles with radii c_1 times larger than their correct proportionate values, so that you can see them clearly. Find a good value for c_1 that makes the planets visible. You will also need to find a good radius for the Sun. Choose any value that gives a clear visualization. (It doesn't work to scale the radius of the Sun by the same factor you use for the planets, because it will come out looking much too large. So just use whatever works.) For added realism, you may also want to make your circles different colors. For instance, Earth could be blue and the Sun could be yellow.

b) The motion of the planets as they move around the Sun (by making the circles move). In the interests of alleviating boredom, construct your program so that time in your animation runs a factor of c_2 faster than real time. Find a good value for c_2 that makes the motion of the orbits easily visible but not unreasonably fast. Make use of the time delay argument of the `draw` function to make your animation run smoothly.

Hint: You could define individual `circle` variables for each planet, but it may be more convenient to store them in an array or a list. You can append `circle` variables to a list just as you would any other variable, or you can create an array of type `circle` with

```
from qdraw import circle
planet = empty(nplanets,circle)
```

In other words, `circle` works as both the name of the function that creates a circle and as the type of the variable it creates.

Chapter summary

- **Graphs** can be produced in Python using the `matplotlib` package, and specifically the module `pyplot`.
- This package includes the functions `plot` for making normal xy graphs, `scatter` for making scatter plots, and `imshow` for making density plots. A fourth function, `show`, displays the finished graph on the screen.
- There are variety of other functions that allow you to modify the appearance of your graphs, including functions for setting the scale on the x and y axes, adding labels to the axes, and changing the color scheme.
- The package `qdraw`, specially written for this book, provides a way to make simple **diagrams** and **animations** of physical systems. It provides functions that draw various objects, such as circles, squares, polygons, and lines, on the screen.
- Once drawn, objects can be moved around with functions that change their location or orientation. Moves only become visible once an additional function `draw` is called. By a sequence of moves, each followed by `draw`, one can then animate the objects on the screen to visualize the behavior of the system.

Further exercises

3.8 Deterministic chaos and the Feigenbaum plot: One of the most famous examples of the phenomenon of chaos is the *logistic map*, defined by the equation

$$x' = rx(1-x).$$

For a given value of the constant r you take a value of x—say $x = \frac{1}{2}$—and you feed it into the right-hand side of this equation, which gives you a value of x'. Then you take that value and feed it back in on the right-hand side again, which gives you another value, and so forth. This is an *iterative map*. You keep doing the same operation over and over on the values of x, and one of three things happens:

1. The value settles down to a fixed number and stays there. This is called a *fixed point*. For instance, $x = 0$ is always a fixed point of the logistic map. (You put $x = 0$ on the right-hand side and you get $x' = 0$ on the left.)
2. It does not settle down to a single value, but it settles down into a periodic pattern, rotating around a set of values, such as say four values, repeating them in sequence. This is called a *limit cycle*.
3. It goes crazy. It generates a seemingly random sequence of numbers that appear to have no rhyme or reason to them at all. This is *deterministic chaos*. "Chaos" because it really does look chaotic, and "deterministic" because even though the values look random,

they're not. They are clearly entirely predictable, because they are given to you by one simple equation. The behavior is *determined*, although it may not look like it.

Write a program that calculates and displays the behavior of the logistic map. Here is what you need to do. For a given value of r, start with $x = \frac{1}{2}$ and iterate the logistic map equation a thousand times. That will give it a chance to settle down to a fixed point or limit cycle if it is going to. Then run for another thousand iterations and plot the points (r, x) on a graph where the horizontal axis is r and the vertical axis is x. You can either use the `plot` function with the options `"ko"` or `"k."` to draw a graph with dots, one for each point, or you can use the `scatter` function to draw a scatter plot (which always uses dots). Repeat the whole calculation for values of r from 1 to 4 in steps of 0.01, plotting the dots for all values of r on the same figure and then finally using the function `show` once to display the complete figure.

Your program should generate a distinctive plot that looks like a tree bent over on its side. This famous picture is called the *Feigenbaum plot*, after its discoverer Mitchell Feigenbaum, or sometimes the *figtree plot*, a play on the fact that it looks like a tree and Feigenbaum means "figtree" in German.[10]

Give answers to the following questions:

a) For a given value of r, what would a fixed point look like on the Feigenbaum plot? How about a limit cycle? And what would chaos look like?

b) Based on your plot, at what value of r does the system move from orderly behavior (fixed points or limit cycles) to chaotic behavior? This point is sometimes called the "edge of chaos."

The logistic map is a very simple mathematical system, but deterministic chaos is seen in many more complex physical systems also, including especially fluid dynamics and the weather. Because of its apparently random nature, the behavior of chaotic systems is difficult to predict and strongly affected by small perturbations in initial conditions or parameter values. You have probably heard of the classic exemplar of chaos in weather systems, the *butterfly effect*, which was popularized by physicist Edward Lorenz in 1972 when he gave a lecture to the American Association for the Advancement of Science entitled, "Does the flap of a butterfly's wings in Brazil set off a tornado in Texas?"[11]

3.9 Hydrogen wavefunction: In suitable units the (spatial part of the) electronic wavefunction of the 2p atomic level of hydrogen is

$$\psi(x, y, z) = z(2 - r)\, \mathrm{e}^{-r},$$

[10] There is another approach for computing the Feigenbaum plot, which is neater and faster, making use of Python's ability to perform arithmetic with entire arrays. You could create an array `r` with one element containing each distinct value of r you want to investigate: `[1.0, 1.01, 1.02, ...]`. Then create another array `x` of the same size to hold the corresponding values of x, which should all be initially set to 0.5. Then an iteration of the logistic map can be performed for all values of r at once with a statement of the form `x = r*x*(1-x)`. Because of the speed with which Python can perform calculations on arrays, this method should be significantly faster than the more basic method above.

[11] Although arguably the first person to suggest the butterfly effect was not a physicist at all, but the science fiction writer Ray Bradbury in his famous 1952 short story *A Sound of Thunder*, in which a time traveler's careless destruction of a butterfly during a tourist trip to the Jurassic era changes the course of history.

where $r = \sqrt{x^2 + y^2 + z^2}$.

a) Write a user-defined function to return the value of $\psi(x, y, z)$ for arbitrary x, y, z.

b) Use your function to make a density plot of the probability density $|\psi|^2$ of the electron in the xz plane, for values of x and z between -2 and 2.

3.10 The Mandelbrot set: The Mandelbrot set, named after its discoverer, French mathematician Benoît Mandelbrot, is a *fractal*, an infinitely ramified mathematical object that contains structure within structure as deep as we care to look. The definition of the Mandelbrot set is in terms of complex numbers as follows.

Consider the equation

$$z' = z^2 + c,$$

where z is a complex number and c is a complex constant. For any given value of c this equation turns an input number z into an output number z'. The definition of the Mandelbrot set involves the repeated iteration of this equation: we take an initial starting value of z and feed it into the equation to get a new value z'. Then we take that value and feed it in again to get another value, and so forth. The Mandelbrot set is the set of points in the complex plane that satisfies the following definition:

> *For a given complex value of c, start with $z = 0$ and iterate repeatedly. If the magnitude $|z|$ of the resulting value is ever greater than 2, then the point in the complex plane at position c is* not *in the Mandelbrot set, otherwise it is in the set.*

In order to use this definition one would, in principle, have to iterate infinitely many times to prove that a point is in the Mandelbrot set, since a point is in the set only if the iteration never passes $|z| = 2$ ever. In practice, however, one usually just performs some large number of iterations, say 100, and if $|z|$ has not exceeded 2 by that point then we call that good enough.

Write a program to make an image of the Mandelbrot set by performing the iteration for all values of $c = x + \mathrm{i}y$ on an $N \times N$ grid spanning the region where $-2 \le x \le 2$ and $-2 \le y \le 2$. Make a density plot in which grid points inside the Mandelbrot set are colored black and those outside are colored white. The Mandelbrot set has a very distinctive shape that looks something like a beetle with a long snout—you'll know it when you see it.

Hint: You will probably find it useful to start off with quite a coarse grid, i.e., with a small value of N—perhaps $N = 100$—so that your program runs quickly while you are testing it. Once you are sure it is working correctly, increase the value of N to produce a final high-quality image of the shape of the set.

If you are feeling enthusiastic, here is another variant of the same exercise that can produce amazing looking pictures. Instead of coloring points just black or white, color points according to the number of iterations of the equation before $|z|$ becomes greater than 2 (or the maximum number of iterations if $|z|$ never becomes greater than 2). If you use one of the more colorful color schemes Python provides for density plots, such as the `viridis` or `jet` schemes, you can make some spectacular images this way. Another interesting variant is to color according to the logarithm of the number of iterations, which helps reveal some of the finer structure outside the set.

3.11 The Spirograph: The Spirograph is a classic mechanical toy, invented in the 1960s, that makes geometric drawings. A plastic ring about 15 or 20 cm across is pinned to a piece of paper. The ring has teeth around its inner rim and a small cog wheel rolls around inside this

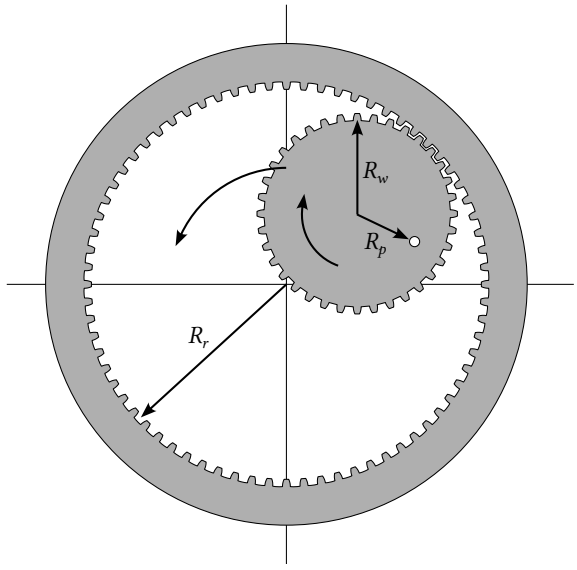


Figure 3.11: The Spirograph. A Spirograph has a toothed wheel that rolls inside a toothed ring. The wheel has a small hole in it, through which one pokes a pen and the pen traces an elaborate flower-like pattern on the paper underneath as it moves.

rim, meshing with the teeth and turning as it goes—see Fig. 3.11. The cog wheel has a hole in it, through which one can stick the tip of a ball-point pen, which draws a trail across the paper as you push the wheel around in circles. The result is a pleasing flower-like pattern, whose details can be adjusted by changing the size of the ring or the wheel. In this exercise, you will write a program to create an animation of the motion of the Spirograph and the pattern it generates.

Suppose the ring of the Spirograph is centered at the origin and its inner rim has radius R_r, as shown in Fig. 3.11. If the wheel has radius R_w, then the distance from the origin to the center of the wheel is $R_r - R_w$ and when the wheel has rolled an angle θ around the rim the position x_w, y_w of the center is given by

$$x_w = (R_r - R_w)\cos\theta, \qquad y_w = (R_r - R_w)\sin\theta.$$

The distance traveled by the wheel along the inside of the rim is θR_r, so the angle ϕ turned by the wheel as it rolls is $\phi = -\theta R_r / R_w$, with the minus sign indicating that the wheel turns in the opposite direction to its movement around the rim. If the distance between the center of the wheel and the pen hole is R_p, then the position of the pen hole is given by

$$x_p = x_w + R_p\cos\phi, \qquad y_p = y_w + R_p\sin\phi.$$

a) Write a program that makes an animation showing the stationary ring, the wheel moving around the inner rim of the ring, and pen hole as it moves, using a circle for each one, with $R_r = 0.83$, $R_w = 0.4$, and $R_p = 0.35$ (in arbitrary units).

b) Remove the circles representing the ring and the wheel from your animation, keeping only the one for the pen hole, and add a trail to represent the line drawn by the pen, so you can see what pattern it generates. (Hint: If you specify no length parameter for the trail it will have no length limit and will record the entire path of the pen hole as it moves.)

c) Vary the values of the three radii R_r, R_w, and R_p and find at least two more settings that produce interesting patterns.