Computational Physics, 2nd edition

Exercises for Chapter 3

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 in 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

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

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 $0 \le \theta \le 24\pi$.

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

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).

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.

Exercise 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

$$x' = rx(1-x). \tag{1}$$

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 a *iterative map*. You keep doing the same operation over and over on your value 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 doesn't 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 over and over. 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're 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's 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's 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, of 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 onto 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.

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 outside conditions. You've 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?" (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.)

Comment: 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.

Exercise 3.9: Hydrogen wavefunction:

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

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

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.

Exercise 3.10: The Mandelbrot set

The Mandelbrot set, named after its discoverer, the French mathematician Benoît Mandelbrot, is a *fractal*, an infinitely ramified mathematical object that contains structure within 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 $|z|$ hasn't 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 $c = x + iy$ 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.

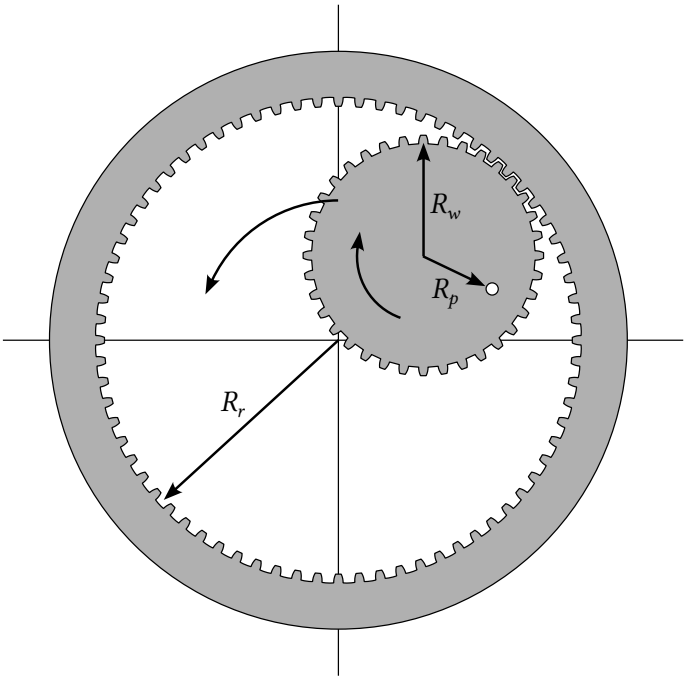
Hint: You will probably find it useful to start off with quite a coarse grid, i.e., with a small value of N—perhaps $N = 100$—so that your program runs quickly while you are testing it. Once you are sure

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 "`hot`" 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.

Exercise 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 rim, meshing with the teeth and turning as it goes:



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 the figure above. 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.