COMPUTATIONAL PHYSICS

EXERCISES FOR CHAPTER 2

Exercise 2.1: Another ball dropped from a tower

A ball is dropped from a tower of height h with initial velocity zero. Write a program that asks the user to enter the height in meters of the tower and then calculates and prints the time the ball takes until it hits the ground, ignoring air resistance. Use your program to calculate the time for a ball dropped from a 100 m high tower.

Exercise 2.2: Altitude of a satellite

A satellite is to be launched into a circular orbit around the Earth so that it orbits the planet once every T seconds.

a) Show that the altitude h above the Earth's surface that the satellite must have is

$$h = \left(\frac{GMT^2}{4\pi^2}\right)^{1/3} - R,$$

where $G = 6.67 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$ is Newton's gravitational constant, $M = 5.97 \times 10^{24}\,\mathrm{kg}$ is the mass of the Earth, and $R = 6371\,\mathrm{km}$ is its radius.

b) Write a program that asks the user to enter the desired value of T and then calculates and prints out the correct altitude in meters.

c) Use your program to calculate the altitudes of satellites that orbit the Earth once a day (so-called "geosynchronous" orbit), once every 90 minutes, and once every 45 minutes. What do you conclude from the last of these calculations?

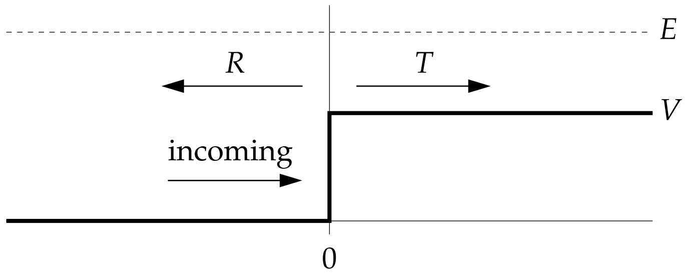
d) Technically a geosynchronous satellite is one that orbits the Earth once per *sidereal day*, which is 23.93 hours, not 24 hours. Why is this? And how much difference will it make to the altitude of the satellite?

Exercise 2.3: Write a program to perform the inverse operation to that of Example 2.2. That is, ask the user for the Cartesian coordinates x, y of a point in two-dimensional space, and calculate and print the corresponding polar coordinates, with the angle θ given in degrees.

Exercise 2.4: A spaceship travels from Earth in a straight line at relativistic speed v to another planet x light years away. Write a program to ask the user for the value of x and the speed v as a fraction of the speed of light c, then print out the time in years that the spaceship takes to reach its destination (a) in the rest frame of an observer on Earth and (b) as perceived by a passenger on board the ship. Use your program to calculate the answers for a planet 10 light years away with $v = 0.99c$.

Exercise 2.5: Quantum potential step

A well-known quantum mechanics problem involves a particle of mass m that encounters a one-dimensional potential step, like this:



The particle with initial kinetic energy E and wavevector $k_1 = \sqrt{2mE}/\hbar$ enters from the left and encounters a sudden jump in potential energy of height V at position $x = 0$. By solving the Schrödinger equation, one can show that when $E > V$ the particle may either (a) pass the step, in which case it has a lower kinetic energy of $E - V$ on the other side and a correspondingly smaller wavevector of $k_2 = \sqrt{2m(E-V)}/\hbar$, or (b) it may be reflected, keeping all of its kinetic energy and an unchanged wavevector but moving in the opposite direction. The probabilities T and R for transmission and reflection are given by

$$T = \frac{4k_1k_2}{(k_1+k_2)^2}, \qquad R = \left(\frac{k_1-k_2}{k_1+k_2}\right)^2.$$

Suppose we have a particle with mass equal to the electron mass $m = 9.11 \times 10^{-31}$ kg and energy 10 eV encountering a potential step of height 9 eV. Write a Python program to compute and print out the transmission and reflection probabilities using the formulas above.

Exercise 2.6: Planetary orbits

The orbit in space of one body around another, such as a planet around the Sun, need not be circular. In general it takes the form of an ellipse, with the body sometimes closer in and sometimes further out. If you are given the distance ℓ_1 of closest approach that a planet makes to the Sun, also called its *perihelion*, and its linear velocity v_1 at perihelion, then any other property of the orbit can be calculated from these two as follows.

a) Kepler's second law tells us that the distance ℓ_2 and velocity v_2 of the planet at its most distant point, or *aphelion*, satisfy $\ell_2 v_2 = \ell_1 v_1$. At the same time the total energy, kinetic plus gravitational, of a planet with velocity v and distance r from the Sun is given by

$$E = \tfrac{1}{2}mv^2 - G\frac{mM}{r},$$

where m is the planet's mass, $M = 1.9891 \times 10^{30}$ kg is the mass of the Sun, and $G = 6.6738 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$ is Newton's gravitational constant. Given that energy must be conserved, show that v_2 is the smaller root of the quadratic equation

$$v_2^2 - \frac{2GM}{v_1\ell_1}v_2 - \left[v_1^2 - \frac{2GM}{\ell_1}\right] = 0.$$

Once we have v_2 we can calculate ℓ_2 using the relation $\ell_2 = \ell_1 v_1 / v_2$.

b) Given the values of v_1, ℓ_1, and ℓ_2, other parameters of the orbit are given by simple formulas can that be derived from Kepler's laws and the fact that the orbit is an ellipse:

$$\begin{aligned}
\text{Semi-major axis:}\quad & a = \tfrac{1}{2}(\ell_1 + \ell_2), \\
\text{Semi-minor axis:}\quad & b = \sqrt{\ell_1 \ell_2}, \\
\text{Orbital period:}\quad & T = \frac{2\pi ab}{\ell_1 v_1}, \\
\text{Orbital eccentricity:}\quad & e = \frac{\ell_2 - \ell_1}{\ell_2 + \ell_1}.
\end{aligned}$$

Write a program that asks the user to enter the distance to the Sun and velocity at perihelion, then calculates and prints the quantities ℓ_2, v_2, T, and e.

c) Test your program by having it calculate the properties of the orbits of the Earth (for which $\ell_1 = 1.4710 \times 10^{11}\,\text{m}$ and $v_1 = 3.0287 \times 10^4\,\text{m}\,\text{s}^{-1}$) and Halley's comet ($\ell_1 = 8.7830 \times 10^{10}\,\text{m}$ and $v_1 = 5.4529 \times 10^4\,\text{m}\,\text{s}^{-1}$). Among other things, you should find that the orbital period of the Earth is one year and that of Halley's comet is about 76 years.

Exercise 2.7: Catalan numbers

The Catalan numbers C_n are a sequence of integers $1, 1, 2, 5, 14, 42, 132\ldots$ that play an important role in quantum mechanics and the theory of disordered systems. (They were central to Eugene Wigner's proof of the so-called semicircle law.) They are given by

$$C_0 = 1, \qquad C_{n+1} = \frac{4n+2}{n+2}\, C_n.$$

Write a program that prints in increasing order all Catalan numbers less than or equal to one billion.

Exercise 2.8: Suppose arrays `a` and `b` are defined as follows:

```
from numpy import array
a = array([1,2,3,4],int)
b = array([2,4,6,8],int)
```

What will the computer print upon executing the following lines? (Try to work out the answer before trying it on the computer.)

a) `print(b/a+1)`
b) `print(b/(a+1))`
c) `print(1/a)`

Exercise 2.9: The Madelung constant

In condensed matter physics the Madelung constant gives the total electric potential felt by an atom in a solid. It depends on the charges on the other atoms nearby and their locations.

Consider for instance solid sodium chloride—table salt. The sodium chloride crystal has atoms arranged on a cubic lattice, but with alternating sodium and chlorine atoms, the sodium ones having a single positive charge $+e$ and the chlorine ones a single negative charge $-e$, where e is the charge on the electron. If we label each position on the lattice by three integer coordinates (i, j, k), then the sodium atoms fall at positions where $i + j + k$ is even, and the chlorine atoms at positions where $i + j + k$ is odd.

Consider a sodium atom at the origin, $i = j = k = 0$, and let us calculate the Madelung constant. If the spacing of atoms on the lattice is a, then the distance from the origin to the atom at position (i, j, k) is

$$\sqrt{(ia)^2 + (ja)^2 + (ka)^2} = a\sqrt{i^2 + j^2 + k^2},$$

and the potential at the origin created by such an atom is

$$V(i, j, k) = \pm \frac{e}{4\pi\epsilon_0 a\sqrt{i^2 + j^2 + k^2}},$$

with ϵ_0 being the permittivity of the vacuum and the sign of the expression depending on whether $i + j + k$ is even or odd. The total potential felt by the sodium atom is then the sum of this quantity over all other atoms. Let us assume a cubic box around the sodium at the origin, with L atoms in all directions. Then

$$V_{\text{total}} = \sum_{\substack{i,j,k=-L \\ \text{not } i=j=k=0}}^{L} V(i, j, k) = \frac{e}{4\pi\epsilon_0 a} M,$$

where M is the Madelung constant, at least approximately—technically the Madelung constant is the value of M when $L \to \infty$, but one can get a good approximation just by using a large value of L.

Write a program to calculate and print the Madelung constant for sodium chloride. Use as large a value of L as you can, while still having your program run in reasonable time—say in a minute or less.

Exercise 2.10: The semi-empirical mass formula

In nuclear physics, the semi-empirical mass formula is a formula for calculating the approximate nuclear binding energy B of an atomic nucleus with atomic number Z and mass number A:

$$B = a_1 A - a_2 A^{2/3} - a_3 \frac{Z^2}{A^{1/3}} - a_4 \frac{(A - 2Z)^2}{A} + \frac{a_5}{A^{1/2}},$$

where, in units of millions of electron volts, the constants are $a_1 = 15.67$, $a_2 = 17.23$, $a_3 = 0.75$, $a_4 = 93.2$, and

$$a_5 = \begin{cases} 0 & \text{if } A \text{ is odd,} \\ 12.0 & \text{if } A \text{ and } Z \text{ are both even,} \\ -12.0 & \text{if } A \text{ is even and } Z \text{ is odd.} \end{cases}$$

a) Write a program that takes as its input the values of A and Z, and prints out the binding energy for the corresponding atom. Use your program to find the binding energy of an atom with $A = 58$ and $Z = 28$. (Hint: The correct answer is around 490 MeV.)

b) Modify your program to print out not the total binding energy B, but the binding energy per nucleon, which is B/A.

c) Now modify your program so that it takes as input just a single value of the atomic number Z and then goes through all values of A from $A = Z$ to $A = 3Z$, to find the one that has the largest binding energy per nucleon. This is the most stable nucleus with the given atomic number. Have your program print out the value of A for this most stable nucleus and the value of the binding energy per nucleon.

d) Modify your program again so that, instead of taking Z as input, it runs through all values of Z from 1 to 100 and prints out the most stable value of A for each one. At what value of Z does the maximum binding energy per nucleon occur? (The true answer, in real life, is $Z = 28$, which is nickel. You should find that the semi-empirical mass formula gets the answer roughly right, but not exactly.)

Exercise 2.11: Binomial coefficients

The binomial coefficient $\binom{n}{k}$ is an integer equal to

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n \times (n-1) \times (n-2) \times \ldots \times (n-k+1)}{1 \times 2 \times \ldots \times k}$$

when $k \geq 1$, or $\binom{n}{0} = 1$ when $k = 0$.

a) Using this form for the binomial coefficient, write a user-defined function `binomial(n,k)` that calculates the binomial coefficient for given n and k. Make sure your function returns the answer in the form of an integer (not a float) and gives the correct value of 1 for the case where $k = 0$.

b) Using your function write a program to print out the first 20 lines of "Pascal's triangle." The nth line of Pascal's triangle contains $n + 1$ numbers, which are the coefficients $\binom{n}{0}$, $\binom{n}{1}$, and so on up to $\binom{n}{n}$. Thus the first few lines are

```
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

c) The probability that an unbiased coin, tossed n times, will come up heads k times is $\binom{n}{k}/2^n$. Write a program to calculate (a) the total probability that a coin tossed 100 times comes up heads exactly 60 times, and (b) the probability that it comes up heads 60 or more times.

Exercise 2.12: Prime numbers

The program in Example 2.8 is not a very efficient way of calculating prime numbers: it checks each number to see if it is divisible by any number less than it. We can develop a much faster program for prime numbers by making use of the following observations:

a) A number n is prime if it has no prime factors less than n. Hence we only need to check if it is divisible by other primes.

b) If a number n is non-prime, having a factor r, then $n = rs$, where s is also a factor. If $r \geq \sqrt{n}$ then $n = rs \geq \sqrt{n}s$, which implies that $s \leq \sqrt{n}$. In other words, any non-prime must have factors, and hence also prime factors, less than or equal to $\sqrt{n}$. Thus to determine if a number is prime we have to check its prime factors only up to and including $\sqrt{n}$—if there are none then the number is prime.

c) If we find even a single prime factor less than $\sqrt{n}$ then we know that the number is non-prime, and hence there is no need to check any further—we can abandon this number and move on to something else.

Write a Python program that finds all the primes up to ten thousand. Create a list to store the primes, which starts out with just the one prime number 2 in it. Then for each number n from 3 to 10 000 check whether the number is divisible by any of the primes in the list up to and including $\sqrt{n}$. As soon as you find a single prime factor you can stop checking the rest of them—you know n is not a prime. If you find no prime factors $\sqrt{n}$ or less then n is prime and you should add it to the list. You can print out the list all in one go at the end of the program, or you can print out the individual numbers as you find them.

Exercise 2.13: Recursion

A useful feature of user-defined functions is *recursion*, the ability of a function to call itself. For example, consider the following definition of the factorial $n!$ of a positive integer n:

$$n! = \begin{cases} 1 & \text{if } n = 1, \\ n \times (n-1)! & \text{if } n > 1. \end{cases}$$

This constitutes a complete definition of the factorial which allows us to calculate the value of $n!$ for any positive integer. We can employ this definition directly to create a Python function for factorials, like this:

```
def factorial(n):
    if n==1:
        return 1
    else:
        return n*factorial(n-1)
```

Note how, if n is not equal to 1, the function calls itself to calculate the factorial of $n-1$. This is recursion. If we now say "`print(factorial(5))`" the computer will correctly print the answer 120.

a) We encountered the Catalan numbers C_n previously in Exercise 2.7 on page 46. With just a little rearrangement, the definition given there can be rewritten in the form

$$C_n = \begin{cases} 1 & \text{if } n = 0, \\ \dfrac{4n-2}{n+1} C_{n-1} & \text{if } n > 0. \end{cases}$$

Write a Python function, using recursion, that calculates C_n. Use your function to calculate and print C_{100}.

b) Euclid showed that the greatest common divisor $g(m,n)$ of two nonnegative integers m and n satisfies

$$g(m,n) = \begin{cases} m & \text{if } n = 0, \\ g(n, m \bmod n) & \text{if } n > 0. \end{cases}$$

Write a Python function `g(m,n)` that employs recursion to calculate the greatest common divisor of m and n using this formula. Use your function to calculate and print the greatest common divisor of 108 and 192.

Comparing the calculation of the Catalan numbers in part (a) above with that of Exercise 2.7, we see that it's possible to do the calculation two ways, either directly or using recursion. In most cases, if a quantity can be calculated *without* recursion, then it will be faster to do so, and we normally recommend taking this route if possible. There are some calculations, however, that are essentially impossible (or at least much more difficult) without recursion. We will see some examples later in this book.

COMPUTATIONAL PHYSICS

EXERCISES FOR CHAPTER 3

Exercise 3.1: Plotting experimental data

In the on-line 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"

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

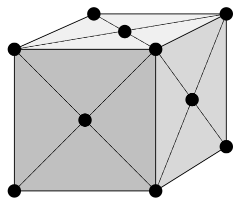
in the range $0 \le \theta \le 24\pi$.

Exercise 3.3: There is a file in the on-line 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 as a starting point, or starting from scratch if you prefer, do the following:

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

b) The face-centered cubic (fcc) lattice, which is the most common lattice in naturally occurring crystals, consists of a cubic lattice with atoms positioned not only at the corners of each cube but also at the center of each face:



Create a visualization of an fcc lattice with a single species of atom (such as occurs in metallic iron, for instance).

Exercise 3.5: 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 facilities provided by the `visual` package, create an animation of the solar system that shows the following:

a) The Sun and planets as spheres 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 spheres 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'll 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'll come out looking much too large. So just use whatever works.) For added realism, you may also want to make your spheres 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 spheres of the planets move). In the interests of alleviating boredom, construct your program so that time in your animation runs a factor of c_2 faster than actual time. Find a good value of c_2 that makes the motion of the orbits easily visible but not unreasonably fast. Make use of the `rate` function to make your animation run smoothly.

Hint: You may find it useful to store the sphere variables representing the planets in an array of the kind described on page 115.

Exercise 3.6: 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 3.7: 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.

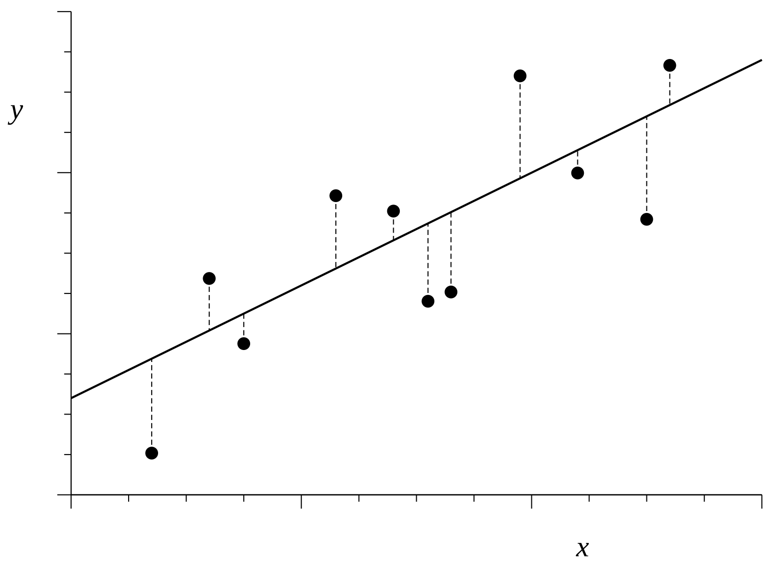
Write a program to make an image of the Mandelbrot set by performing the iteration for all values of $c = x + iy$ on an $N \times N$ grid spanning the region where $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$. Make a density plot in which grid points inside the Mandelbrot set are colored black and those 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 3.8: Least-squares fitting and the photoelectric effect

It's a common situation in physics that an experiment produces data that lies roughly on a straight line, like the dots in this figure:



The solid line here represents the underlying straight-line form, which we usually don't know, and the points representing the measured data lie roughly along the line but don't fall exactly on it, typically because of measurement error.

The straight line can be represented in the familiar form $y = mx + c$ and a frequent question is what the appropriate values of the slope m and intercept c are that correspond to the measured data. Since the data don't fall perfectly on a straight line, there is no perfect answer to such a question, but we can find the straight line that gives the best compromise fit to the data. The standard technique for doing this is the *method of least squares*.

Suppose we make some guess about the parameters m and c for the straight line. We then calculate the vertical distances between the data points and that line, as represented by the short vertical lines in the figure, then we calculate the sum of the squares of those distances, which we denote χ^2. If we have N data points with coordinates (x_i, y_i), then χ^2 is given by

$$\chi^2 = \sum_{i=1}^{N} (mx_i + c - y_i)^2.$$

The least-squares fit of the straight line to the data is the straight line that minimizes this total squared distance from data to line. We find the minimum by differentiating with respect to both m and c and setting the derivatives to zero, which gives

$$m \sum_{i=1}^{N} x_i^2 + c \sum_{i=1}^{N} x_i - \sum_{i=1}^{N} x_i y_i = 0,$$
$$m \sum_{i=1}^{N} x_i + cN - \sum_{i=1}^{N} y_i = 0.$$

For convenience, let us define the following quantities:

$$E_x = \frac{1}{N} \sum_{i=1}^{N} x_i, \qquad E_y = \frac{1}{N} \sum_{i=1}^{N} y_i, \qquad E_{xx} = \frac{1}{N} \sum_{i=1}^{N} x_i^2, \qquad E_{xy} = \frac{1}{N} \sum_{i=1}^{N} x_i y_i,$$

in terms of which our equations can be written

$$mE_{xx} + cE_x = E_{xy}\,,$$
$$mE_x + c = E_y\,.$$

Solving these equations simultaneously for m and c now gives

$$m = \frac{E_{xy} - E_x E_y}{E_{xx} - E_x^2}, \qquad c = \frac{E_{xx}E_y - E_x E_{xy}}{E_{xx} - E_x^2}.$$

These are the equations for the least-squares fit of a straight line to N data points. They tell you the values of m and c for the line that best fits the given data.

a) In the on-line resources you will find a file called `millikan.txt`. The file contains two columns of numbers, giving the x and y coordinates of a set of data points. Write a program to read these data points and make a graph with one dot or circle for each point.

b) Add code to your program, before the part that makes the graph, to calculate the quantities E_x, E_y, E_{xx}, and E_{xy} defined above, and from them calculate and print out the slope m and intercept c of the best-fit line.

c) Now write code that goes through each of the data points in turn and evaluates the quantity $mx_i + c$ using the values of m and c that you calculated. Store these values in a new array or list, and then graph this new array, as a solid line, on the same plot as the original data. You should end up with a plot of the data points plus a straight line that runs through them.

d) The data in the file `millikan.txt` are taken from a historic experiment by Robert Millikan that measured the *photoelectric effect*. When light of an appropriate wavelength is shone on the surface of a metal, the photons in the light can strike conduction electrons in the metal and, sometimes, eject them from the surface into the free space above. The energy of an ejected electron is equal to the energy of the photon that struck it minus a small amount ϕ called the *work function* of the surface, which represents the energy needed to remove an electron from the surface. The energy of a photon is $h\nu$, where h is Planck's constant and ν is the frequency of the light, and we can measure the energy of an ejected electron by measuring the voltage V that is just sufficient to stop the electron moving. Then the voltage, frequency, and work function are related by the equation

$$V = \frac{h}{e}\nu - \phi,$$

where e is the charge on the electron. This equation was first given by Albert Einstein in 1905.

The data in the file `millikan.txt` represent frequencies ν in hertz (first column) and voltages V in volts (second column) from photoelectric measurements of this kind. Using the equation above and the program you wrote, and given that the charge on the electron is 1.602×10^{-19} C, calculate from Millikan's experimental data a value for Planck's constant. Compare your value with the accepted value of the constant, which you can find in books or on-line. You should get a result within a couple of percent of the accepted value.

This calculation is essentially the same as the one that Millikan himself used to determine of the value of Planck's constant, although, lacking a computer, he fitted his straight line to the data by eye. In part for this work, Millikan was awarded the Nobel prize in physics in 1923.

Computational Physics

Exercises for Chapter 4

Exercise 4.1: Write a program to calculate and print the factorial of a number entered by the user. If you wish you can base your program on the user-defined function for factorial given in Section 2.6, but write your program so that it calculates the factorial using *integer* variables, not floating-point ones. Use your program to calculate the factorial of 200.

Now modify your program to use floating-point variables instead and again calculate the factorial of 200. What do you find? Explain.

Exercise 4.2: Quadratic equations

a) Write a program that takes as input three numbers, a, b, and c, and prints out the two solutions to the quadratic equation $ax^2 + bx + c = 0$ using the standard formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Use your program to compute the solutions of $0.001x^2 + 1000x + 0.001 = 0$.

b) There is another way to write the solutions to a quadratic equation. Multiplying top and bottom of the solution above by $-b \mp \sqrt{b^2 - 4ac}$, show that the solutions can also be written as

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}.$$

Add further lines to your program to print out these values in addition to the earlier ones and again use the program to solve $0.001x^2 + 1000x + 0.001 = 0$. What do you see? How do you explain it?

c) Using what you have learned, write a new program that calculates both roots of a quadratic equation accurately in all cases.

This is a good example of how computers don't always work the way you expect them to. If you simply apply the standard formula for the quadratic equation, the computer will sometimes get the wrong answer. In practice the method you have worked out here is the correct way to solve a quadratic equation on a computer, even though it's more complicated than the standard formula. If you were writing a program that involved solving many quadratic equations this method might be a good candidate for a user-defined function: you could put the details of the solution method inside a function to save yourself the trouble of going through it step by step every time you have a new equation to solve.

Exercise 4.3: Calculating derivatives

Suppose we have a function $f(x)$ and we want to calculate its derivative at a point x. We can do that with pencil and paper if we know the mathematical form of the function, or we can do it on the computer by making use of the definition of the derivative:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{\delta\to 0} \frac{f(x+\delta) - f(x)}{\delta}.$$

On the computer we can't actually take the limit as δ goes to zero, but we can get a reasonable approximation just by making δ small.

a) Write a program that defines a function `f(x)` returning the value $x(x-1)$, then calculates the derivative of the function at the point $x = 1$ using the formula above with $\delta = 10^{-2}$. Calculate the true value of the same derivative analytically and compare with the answer your program gives. The two will not agree perfectly. Why not?

b) Repeat the calculation for $\delta = 10^{-4}$, 10^{-6}, 10^{-8}, 10^{-10}, 10^{-12}, and 10^{-14}. You should see that the accuracy of the calculation initially gets better as δ gets smaller, but then gets worse again. Why is this?

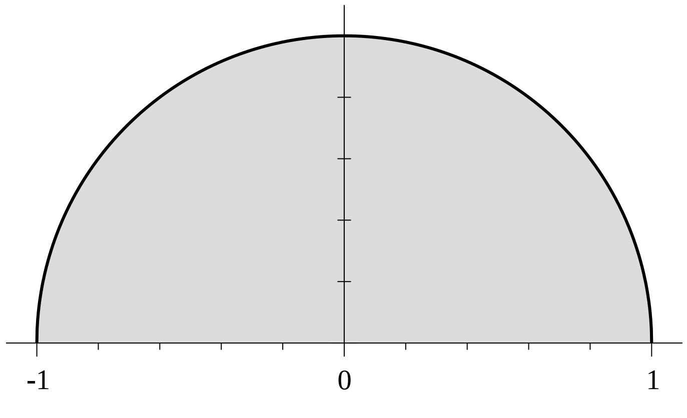
We will look at numerical derivatives in more detail in Section 5.10, where we will study techniques for dealing with these issues and maximizing the accuracy of our calculations.

Exercise 4.4: Calculating integrals

Suppose we want to calculate the value of the integral

$$I = \int_{-1}^{1} \sqrt{1-x^2}\,\mathrm{d}x.$$

The integrand looks like a semicircle of radius 1:



and hence the value of the integral—the area under the curve—must be $\frac{1}{2}\pi = 1.57079632679\ldots$

Alternatively, we can evaluate the integral on the computer by dividing the domain of integration into a large number N of slices of width $h = 2/N$ each and then using the Riemann definition of the integral:

$$I = \lim_{N\to\infty} \sum_{k=1}^{N} h y_k\,,$$

where

$$y_k = \sqrt{1 - x_k^2} \qquad \text{and} \qquad x_k = -1 + hk.$$

We cannot in practice take the limit $N \to \infty$, but we can make a reasonable approximation by just making N large.

a) Write a program to evaluate the integral above with $N = 100$ and compare the result with the exact value. The two will not agree very well, because $N = 100$ is not a sufficiently large number of slices.

b) Increase the value of N to get a more accurate value for the integral. If we require that the program runs in about one second or less, how accurate a value can you get?

Evaluating integrals is a common task in computational physics calculations. We will study techniques for doing integrals in detail in the next chapter. As we will see, there are substantially quicker and more accurate methods than the simple one we have used here.

Computational Physics

Exercises for Chapter 5

Exercise 5.1: In the on-line resources you will find a file called `velocities.txt`, which contains two columns of numbers, the first representing time t in seconds and the second the x-velocity in meters per second of a particle, measured once every second from time $t = 0$ to $t = 100$. The first few lines look like this:

```
0         0
1         0.069478
2         0.137694
3         0.204332
4         0.269083
5         0.331656
```

Write a program to do the following:

a) Read in the data and, using the trapezoidal rule, calculate from them the approximate distance traveled by the particle in the x direction as a function of time. See Section 2.4.3 on page 57 if you want a reminder of how to read data from a file.

b) Extend your program to make a graph that shows, on the same plot, both the original velocity curve and the distance traveled as a function of time.

Exercise 5.2:

a) Write a program to calculate an approximate value for the integral $\int_0^2 (x^4 - 2x + 1)\,\mathrm{d}x$ from Example 5.1, but using Simpson's rule with 10 slices instead of the trapezoidal rule. You may wish to base your program on the trapezoidal rule program on page 142.

b) Run the program and compare your result to the known correct value of 4.4. What is the fractional error on your calculation?

c) Modify the program to use a hundred slices instead, then a thousand. Note the improvement in the result. How do the results compare with those from Example 5.1 for the trapezoidal rule with the same numbers of slices?

Exercise 5.3: Consider the integral

$$E(x) = \int_0^x e^{-t^2}\,\mathrm{d}t.$$

a) Write a program to calculate $E(x)$ for values of x from 0 to 3 in steps of 0.1. Choose for yourself what method you will use for performing the integral and a suitable number of slices.

b) When you are convinced your program is working, extend it further to make a graph of $E(x)$ as a function of x. If you want to remind yourself of how to make a graph, you should consult Section 3.1, starting on page 88.

Note that there is no known way to perform this particular integral analytically, so numerical approaches are the only way forward.

Exercise 5.4: The diffraction limit of a telescope

Our ability to resolve detail in astronomical observations is limited by the diffraction of light in our telescopes. Light from stars can be treated effectively as coming from a point source at infinity. When such light, with wavelength λ, passes through the circular aperture of a telescope (which we'll assume to have unit radius) and is focused by the telescope in the focal plane, it produces not a single dot, but a circular diffraction pattern consisting of central spot surrounded by a series of concentric rings. The intensity of the light in this diffraction pattern is given by

$$I(r) = \left(\frac{J_1(kr)}{kr}\right)^2,$$

where r is the distance in the focal plane from the center of the diffraction pattern, $k = 2\pi/\lambda$, and $J_1(x)$ is a Bessel function. The Bessel functions $J_m(x)$ are given by

$$J_m(x) = \frac{1}{\pi}\int_0^\pi \cos(m\theta - x\sin\theta)\,\mathrm{d}\theta,$$

where m is a nonnegative integer and $x \geq 0$.

a) Write a Python function `J(m,x)` that calculates the value of $J_m(x)$ using Simpson's rule with $N = 1000$ points. Use your function in a program to make a plot, on a single graph, of the Bessel functions J_0, J_1, and J_2 as a function of x from $x = 0$ to $x = 20$.

b) Make a second program that makes a density plot of the intensity of the circular diffraction pattern of a point light source with $\lambda = 500\,\text{nm}$, in a square region of the focal plane, using the formula given above. Your picture should cover values of r from zero up to about $1\,\mu\text{m}$.

Hint 1: You may find it useful to know that $\lim_{x\to0} J_1(x)/x = \frac{1}{2}$. Hint 2: The central spot in the diffraction pattern is so bright that it may be difficult to see the rings around it on the computer screen. If you run into this problem a simple way to deal with it is to use one of the other color schemes for density plots described in Section 3.3. The "`hot`" scheme works well. For a more sophisticated solution to the problem, the `imshow` function has an additional argument `vmax` that allows you to set the value that corresponds to the brightest point in the plot. For instance, if you say "`imshow(x,vmax=0.1)`", then elements in `x` with value 0.1, or any greater value, will produce the brightest (most positive) color on the screen. By lowering the `vmax` value, you can reduce the total range of values between the minimum and maximum brightness, and hence increase the sensitivity of the plot, making subtle details visible. (There is also a `vmin` argument that can be used to set the value that corresponds to the dimmest (most negative) color.) For this exercise a value of `vmax=0.01` appears to work well.

Exercise 5.5: Error on Simpson's rule

Following the same line of argument that led to Eq. (5.28), show that the error on an integral evaluated using Simpson's rule is given, to leading order in h, by Eq. (5.29).

Exercise 5.6: Write a program, or modify an earlier one, to once more calculate the value of the integral $\int_0^2 (x^4 - 2x + 1)\,\mathrm{d}x$ from Example (5.28), using the trapezoidal rule with 20 slices, but this time have the program also print an estimate of the error on the result, calculated using the method of Eq. (5.28). To do this you will need to evaluate the integral twice, once with $N_1 = 10$ slices and then again with $N_2 = 20$ slices. Then Eq. (5.28) gives the error. How does the error calculated in this manner compare with a direct computation of the error as the difference between your value for the integral and the true value of 4.4? Why do the two not agree perfectly?

Exercise 5.7: Consider the integral

$$I = \int_0^1 \sin^2 \sqrt{100x}\,\mathrm{d}x$$

a) Write a program that uses the adaptive trapezoidal rule method of Section 5.3 and Eq. (5.34) to calculate the value of this integral to an approximate accuracy of $\epsilon = 10^{-6}$ (i.e., correct to six digits after the decimal point). Start with one single integration slice and work up from there to two, four, eight, and so forth. Have your program print out the number of slices, its estimate of the integral, and its estimate of the error on the integral, for each value of the number of slices N, until the target accuracy is reached. (Hint: You should find the result is around $I = 0.45$.)

b) Now modify your program to evaluate the same integral using the Romberg integration technique described in this section. Have your program print out a triangular table of values, as on page 161, of all the Romberg estimates of the integral. Calculate the error on your estimates using Eq. (5.49) and again continue the calculation until you reach an accuracy of $\epsilon = 10^{-6}$. You should find that the Romberg method reaches the required accuracy considerably faster than the trapezoidal rule alone.

Exercise 5.8: Write a program that uses the adaptive Simpson's rule method of Section 5.3 and Eqs. (5.35) to (5.39) to calculate the same integral as in Exercise 5.7, again to an approximate accuracy of $\epsilon = 10^{-6}$. Starting this time with two integration slices, work up from there to four, eight, and so forth, printing out the results at each step until the required accuracy is reached. You should find you reach that accuracy for a significantly smaller number of slices than with the trapezoidal rule calculation in part (a) of Exercise 5.7, but a somewhat larger number than with the Romberg integration of part (b).

Exercise 5.9: Heat capacity of a solid

Debye's theory of solids gives the heat capacity of a solid at temperature T to be

$$C_V = 9V\rho k_B \left(\frac{T}{\theta_D}\right)^3 \int_0^{\theta_D/T} \frac{x^4 \mathrm{e}^x}{(\mathrm{e}^x - 1)^2}\,\mathrm{d}x,$$

where V is the volume of the solid, ρ is the number density of atoms, k_B is Boltzmann's constant, and θ_D is the so-called *Debye temperature*, a property of solids that depends on their density and speed of sound.

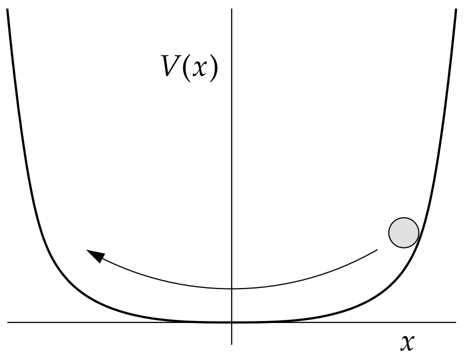
a) Write a Python function `cv(T)` that calculates C_V for a given value of the temperature, for a sample consisting of 1000 cubic centimeters of solid aluminum, which has a number density of $\rho = 6.022 \times 10^{28}\,\mathrm{m}^{-3}$ and a Debye temperature of $\theta_D = 428\,\mathrm{K}$. Use Gaussian quadrature to evaluate the integral, with $N = 50$ sample points.

b) Use your function to make a graph of the heat capacity as a function of temperature from $T = 5\,\mathrm{K}$ to $T = 500\,\mathrm{K}$.

Exercise 5.10: Period of an anharmonic oscillator

The simple harmonic oscillator crops up in many places. Its behavior can be studied readily using analytic methods and it has the important property that its period of oscillation is a constant, independent of its amplitude, making it useful, for instance, for keeping time in watches and clocks. Frequently in physics, however, we also come across anharmonic oscillators, whose period varies with amplitude and whose behavior cannot usually be calculated analytically.

A general classical oscillator can be thought of as a particle in a concave potential well. When disturbed, the particle will rock back and forth in the well:



The harmonic oscillator corresponds to a quadratic potential $V(x) \propto x^2$. Any other form gives an anharmonic oscillator. (Thus there are many different kinds of anharmonic oscillator, depending on the exact form of the potential.)

One way to calculate the motion of an oscillator is to write down the equation for the conservation of energy in the system. If the particle has mass m and position x, then the total energy is equal to the sum of the kinetic and potential energies thus:

$$E = \tfrac{1}{2}m\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2 + V(x).$$

Since the energy must be constant over time, this equation is effectively a (nonlinear) differential equation linking x and t.

Let us assume that the potential $V(x)$ is symmetric about $x = 0$ and let us set our anharmonic oscillator going with amplitude a. That is, at $t = 0$ we release it from rest at position

$x = a$ and it swings back towards the origin. Then at $t = 0$ we have $\mathrm{d}x/\mathrm{d}t = 0$ and the equation above reads $E = V(a)$, which gives us the total energy of the particle in terms of the amplitude.

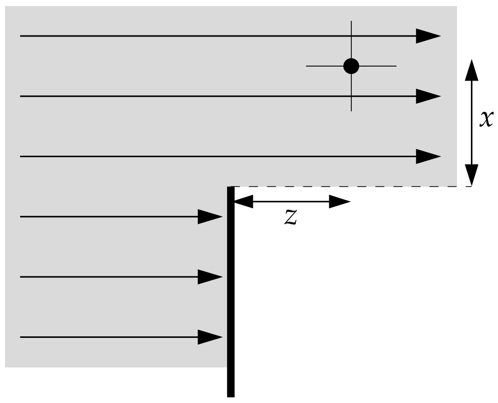
a) When the particle reaches the origin for the first time, it has gone through one quarter of a period of the oscillator. By rearranging the equation above for $\mathrm{d}x/\mathrm{d}t$ and then integrating with respect to t from 0 to $\frac{1}{4}T$, show that the period T is given by

$$T = \sqrt{8m} \int_0^a \frac{\mathrm{d}x}{\sqrt{V(a) - V(x)}}.$$

b) Suppose the potential is $V(x) = x^4$ and the mass of the particle is $m = 1$. Write a Python function that calculates the period of the oscillator for given amplitude a using Gaussian quadrature with $N = 20$ points, then use your function to make a graph of the period for amplitudes ranging from $a = 0$ to $a = 2$.

c) You should find that the oscillator gets faster as the amplitude increases, even though the particle has further to travel for larger amplitude. And you should find that the period diverges as the amplitude goes to zero. How do you explain these results?

Exercise 5.11: Suppose a plane wave, such as light or a sound wave, is blocked by an object with a straight edge, represented by the solid line at the bottom of this figure:



The wave will be diffracted at the edge and the resulting intensity at the position (x, z) marked by the dot is given by near-field diffraction theory to be

$$I = \frac{I_0}{8}\Big(\big[2C(u) + 1\big]^2 + \big[2S(u) + 1\big]^2\Big),$$

where I_0 is the intensity of the wave before diffraction and

$$u = x\sqrt{\frac{2}{\lambda z}}, \qquad C(u) = \int_0^u \cos \tfrac{1}{2}\pi t^2 \,\mathrm{d}t, \qquad S(u) = \int_0^u \sin \tfrac{1}{2}\pi t^2 \,\mathrm{d}t.$$

Write a program to calculate I/I_0 and make a plot of it as a function of x in the range $-5\,\mathrm{m}$ to $5\,\mathrm{m}$ for the case of a sound wave with wavelength $\lambda = 1\,\mathrm{m}$, measured $z = 3\,\mathrm{m}$ past the straight edge. Calculate the integrals using Gaussian quadrature with $N = 50$ points. You should find significant variation in the intensity of the diffracted sound—enough that you could easily hear the effect if sound were diffracted, say, at the edge of a tall building.

Exercise 5.12: The Stefan–Boltzmann constant

The Planck theory of thermal radiation tells us that in the (angular) frequency interval ω to $\omega + \mathrm{d}\omega$, a black body of unit area radiates electromagnetically an amount of thermal energy per second equal to $I(\omega)\,\mathrm{d}\omega$, where

$$I(\omega) = \frac{\hbar}{4\pi^2 c^2} \frac{\omega^3}{(\mathrm{e}^{\hbar\omega/k_B T} - 1)}.$$

Here $\hbar$ is Planck's constant over 2π, c is the speed of light, and k_B is Boltzmann's constant.

a) Show that the total energy per unit area radiated by a black body is

$$W = \frac{k_B^4 T^4}{4\pi^2 c^2 \hbar^3} \int_0^\infty \frac{x^3}{\mathrm{e}^x - 1}\,\mathrm{d}x.$$

b) Write a program to evaluate the integral in this expression. Explain what method you used, and how accurate you think your answer is.

c) Even before Planck gave his theory of thermal radiation around the turn of the 20th century, it was known that the total energy W given off by a black body per unit area per second followed Stefan's law: $W = \sigma T^4$, where σ is the Stefan–Boltzmann constant. Use your value for the integral above to compute a value for the Stefan–Boltzmann constant (in SI units) to three significant figures. Check your result against the known value, which you can find in books or on-line. You should get good agreement.

Exercise 5.13: Quantum uncertainty in the harmonic oscillator

In units where all the constants are 1, the wavefunction of the nth energy level of the one-dimensional quantum harmonic oscillator—i.e., a spinless point particle in a quadratic potential well—is given by

$$\psi_n(x) = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}}\,\mathrm{e}^{-x^2/2} H_n(x),$$

for $n = 0 \ldots \infty$, where $H_n(x)$ is the nth Hermite polynomial. Hermite polynomials satisfy a relation somewhat similar to that for the Fibonacci numbers, although more complex:

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

The first two Hermite polynomials are $H_0(x) = 1$ and $H_1(x) = 2x$.

a) Write a user-defined function `H(n,x)` that calculates $H_n(x)$ for given x and any integer $n \geq 0$. Use your function to make a plot that shows the harmonic oscillator wavefunctions for $n = 0$, 1, 2, and 3, all on the same graph, in the range $x = -4$ to $x = 4$. Hint: There is a function `factorial` in the `math` package that calculates the factorial of an integer.

b) Make a separate plot of the wavefunction for $n = 30$ from $x = -10$ to $x = 10$. Hint: If your program takes too long to run in this case, then you're doing the calculation wrong—the program should take only a second or so to run.

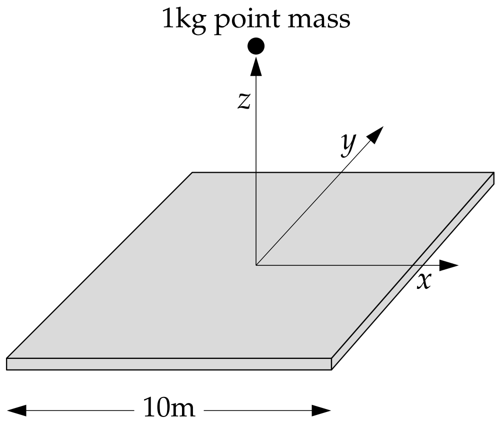
c) The quantum uncertainty of a particle in the nth level of a quantum harmonic oscillator can be quantified by its root-mean-square position $\sqrt{\langle x^2 \rangle}$, where

$$\langle x^2 \rangle = \int_{-\infty}^{\infty} x^2 |\psi_n(x)|^2 \, dx.$$

Write a program that evaluates this integral using Gaussian quadrature on 100 points and then calculates the uncertainty (i.e., the root-mean-square position of the particle) for a given value of n. Use your program to calculate the uncertainty for $n = 5$. You should get an answer in the vicinity of $\sqrt{\langle x^2 \rangle} = 2.3$.

Exercise 5.14: Gravitational pull of a uniform sheet

A uniform square sheet of metal is floating motionless in space:



The sheet is 10 m on a side and of negligible thickness, and it has a mass of 10 metric tonnes.

a) Consider the gravitational force due to the plate felt by a point mass of 1 kg a distance z from the center of the square, in the direction perpendicular to the sheet, as shown above. Show that the component of the force along the z-axis is

$$F_z = G\sigma z \iint_{-L/2}^{L/2} \frac{dx\,dy}{(x^2 + y^2 + z^2)^{3/2}},$$

where $G = 6.674 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$ is Newton's gravitational constant and σ is the mass per unit area of the sheet.

b) Write a program to calculate and plot the force as a function of z from $z = 0$ to $z = 10$ m. For the double integral use (double) Gaussian quadrature, as in Eq. (5.82), with 100 sample points along each axis.

c) You should see a smooth curve, except at very small values of z, where the force should drop off suddenly to zero. This drop is not a real effect, but an artifact of the way we have done the calculation. Explain briefly where this artifact comes from and suggest a strategy to remove it, or at least to decrease its size.

This calculation can thought of as a model for the gravitational pull of a galaxy. Most of the mass in a spiral galaxy (such as our own Milky Way) lies in a thin plane or disk passing through the galactic center, and the gravitational pull exerted by that plane on bodies outside the galaxy can be calculated by just the methods we have employed here.

Exercise 5.15: Create a user-defined function `f(x)` that returns the value $1 + \frac{1}{2}\tanh 2x$, then use a central difference to calculate the derivative of the function in the range $-2 \le x \le 2$. Calculate an analytic formula for the derivative and make a graph with your numerical result and the analytic answer on the same plot. It may help to plot the exact answer as lines and the numerical one as dots. (Hint: In Python the tanh function is found in the `math` package, and it's called simply `tanh`.)

Exercise 5.16: Even when we can find the value of $f(x)$ for any value of x the forward difference can still be more accurate than the central difference for sufficiently large h. For what values of h will the approximation error on the forward difference of Eq. (5.87) be smaller than on the central difference of Eq. (5.95)?

Exercise 5.17: The gamma function

A commonly occurring function in physics calculations is the gamma function $\Gamma(a)$, which is defined by the integral

$$\Gamma(a) = \int_0^\infty x^{a-1} e^{-x}\, dx.$$

There is no closed-form expression for the gamma function, but one can calculate its value for given a by performing the integral above numerically. You have to be careful how you do it, however, if you wish to get an accurate answer.

a) Write a program to make a graph of the value of the integrand $x^{a-1}e^{-x}$ as a function of x from $x = 0$ to $x = 5$, with three separate curves for $a = 2$, 3, and 4, all on the same axes. You should find that the integrand starts at zero, rises to a maximum, and then decays again for each curve.

b) Show analytically that the maximum falls at $x = a - 1$.

c) Most of the area under the integrand falls near the maximum, so to get an accurate value of the gamma function we need to do a good job of this part of the integral. We can change the integral from 0 to ∞ to one over a finite range from 0 to 1 using the change of variables in Eq. (5.67), but this tends to squash the peak towards the edge of the $[0, 1]$ range and does a poor job of evaluating the integral accurately. We can do a better job by making a different change of variables that puts the peak in the middle of the integration range, around $\frac{1}{2}$. We will use the change of variables given in Eq. (5.69), which we repeat here for convenience:

$$z = \frac{x}{c + x}.$$

For what value of x does this change of variables give $z = \frac{1}{2}$? Hence what is the appropriate choice of the parameter c that puts the peak of the integrand for the gamma function at $z = \frac{1}{2}$?

d) Before we can calculate the gamma function, there is another detail we need to attend to. The integrand $x^{a-1}\mathrm{e}^{-x}$ can be difficult to evaluate because the factor x^{a-1} can become very large and the factor e^{-x} very small, causing numerical overflow or underflow, or both, for some values of x. Write $x^{a-1} = \mathrm{e}^{(a-1)\ln x}$ to derive an alternative expression for the integrand that does not suffer from these problems (or at least not so much). Explain why your new expression is better than the old one.

e) Now, using the change of variables above and the value of c you have chosen, write a user-defined function `gamma(a)` to calculate the gamma function for arbitrary argument a. Use whatever integration method you feel is appropriate. Test your function by using it to calculate and print the value of $\Gamma(\frac{3}{2})$, which is known to be equal to $\frac{1}{2}\sqrt{\pi} \simeq 0.886$.

f) For integer values of a it can be shown that $\Gamma(a)$ is equal to the factorial of $a-1$. Use your Python function to calculate $\Gamma(3)$, $\Gamma(6)$, and $\Gamma(10)$. You should get answers closely equal to $2! = 2$, $5! = 120$, and $9! = 362\,880$.

Exercise 5.18: Rearranging Eq. (5.19) into a slightly more conventional form, we have:

$$\int_a^b f(x)\,\mathrm{d}x = h\left[\tfrac{1}{2}f(a) + \tfrac{1}{2}f(b) + \sum_{k=1}^{N-1} f(a+kh)\right] + \tfrac{1}{12}h^2\big[f'(a) - f'(b)\big] + \mathrm{O}(h^4).$$

This result gives a value for the integral on the left which has an error of order h^4—a factor of h^2 better than the error on the trapezoidal rule and as good as Simpson's rule. We can use this formula as a new rule for evaluating integrals, distinct from any of the others we have seen in this chapter. We might call it the "Euler–Maclaurin rule."

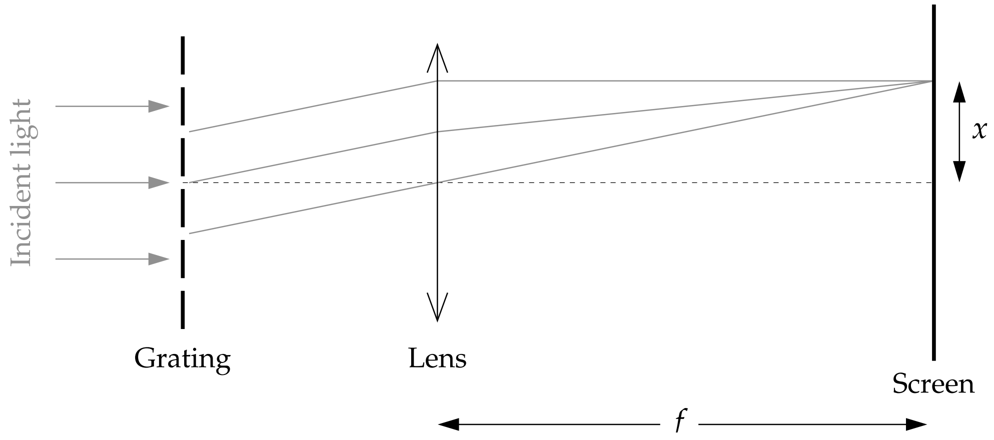
a) Write a program to calculate the value of the integral $\int_0^2 (x^4 - 2x + 1)\,\mathrm{d}x$ using this formula. (This is the same integral that we studied in Example 5.1, whose true value is 4.4.) The order-h term in the formula is just the ordinary trapezoidal rule; the h^2 term involves the derivatives $f'(a)$ and $f'(b)$, which you should evaluate using central differences, centered on a and b respectively. Note that the size of the interval you use for calculating the central differences does not have to equal the value of h used in the trapezoidal rule part of the calculation. An interval of about 10^{-5} gives good values for the central differences.

   Use your program to evaluate the integral with $N = 10$ slices and compare the accuracy of the result with that obtained from the trapezoidal rule alone with the same number of slices.

b) Good though it is, this integration method is not much used in practice. Suggest a reason why not.

Exercise 5.19: Diffraction gratings

Light with wavelength λ is incident on a diffraction grating of total width w, gets diffracted, is focused with a lens of focal length f, and falls on a screen:



Theory tells us that the intensity of the diffraction pattern on the screen, a distance x from the central axis of the system, is given by

$$I(x) = \left| \int_{-w/2}^{w/2} \sqrt{q(u)}\, \mathrm{e}^{\mathrm{i}2\pi x u/\lambda f}\, \mathrm{d}u \right|^2,$$

where $q(u)$ is the intensity transmission function of the diffraction grating at a distance u from the central axis, i.e., the fraction of the incident light that the grating lets through.

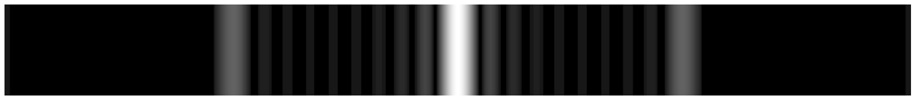
a) Consider a grating with transmission function $q(u) = \sin^2 \alpha u$. What is the separation of the "slits" in this grating, expressed in terms of α?

b) Write a Python function `q(u)` that returns the transmission function $q(u) = \sin^2 \alpha u$ as above at position u for a grating whose slits have separation $20\,\mu$m.

c) Use your function in a program to calculate and graph the intensity of the diffraction pattern produced by such a grating having ten slits in total, if the incident light has wavelength $\lambda = 500\,$nm. Assume the lens has a focal length of 1 meter and the screen is 10 cm wide. You can use whatever method you think appropriate for doing the integral. Once you've made your choice you'll also need to decide the number of sample points you'll use. What criteria play into this decision?

   Notice that the integrand in the equation for $I(x)$ is complex, so you will have to use complex variables in your program. As mentioned in Section 2.2.5, there is a version of the `math` package for use with complex variables called `cmath`. In particular you may find the `exp` function from `cmath` useful because it can calculate the exponentials of complex arguments.

d) Create a visualization of how the diffraction pattern would look on the screen using a density plot (see Section 3.3). Your plot should look something like this:



e) Modify your program further to make pictures of the diffraction patterns produced by gratings with the following profiles:

i) A transmission profile that obeys $q(u) = \sin^2 \alpha u \sin^2 \beta u$, with α as before and the same total grating width w, and $\beta = \frac{1}{2}\alpha$.

ii) Two "square" slits, meaning slits with 100% transmission through the slit and 0% transmission everywhere else. Calculate the diffraction pattern for non-identical slits, one 10 μm wide and the other 20 μm wide, with a 60 μm gap between the two.

Exercise 5.20: A more advanced adaptive method for the trapezoidal rule

In Section 5.3 we studied an adaptive version of the trapezoidal rule in which the number of steps is increased—and the width h of the slices correspondingly decreased—until the calculation gives a value for the integral accurate to some desired level. Although this method varies h, it still calculates the integral at any individual stage of the process using slices of equal width throughout the domain of integration. In this exercise we look at a more sophisticated form of the trapezoidal rule that uses different step sizes in different parts of the domain, which can be useful particularly for poorly behaved functions that vary rapidly in certain regions but not others. Remarkably, this method is not much more complicated to program than the ones we've already seen, if one knows the right tricks. Here's how the method works.

Suppose we wish to evaluate the integral $I = \int_a^b f(x)\,\mathrm{d}x$ and we want an error of no more than ϵ on our answer. To put that another way, if we divide up the integral into slices of width h then we require an accuracy per slice of

$$h\,\frac{\epsilon}{b-a} = h\delta,$$

where $\delta = \epsilon/(b-a)$ is the target accuracy per unit interval.

We start by evaluating the integral using the trapezoidal rule with just a single slice of width $h_1 = b - a$. Let us call the estimate of the integral from this calculation I_1. Usually I_1 will not be very accurate, but let us ignore that for the moment. Next we make a second estimate I_2 of the integral, again using the trapezoidal rule but now with two slices of width $h_2 = \frac{1}{2}h_1$ each. Equation (5.28) tells us that the error on this second estimate is $\frac{1}{3}(I_2 - I_1)$ to leading order. If this error is smaller than the desired accuracy $h_2\delta$ then our calculation is complete and we need go no further. I_2 is a good enough estimate of the integral.

Most likely, however, this will not be the case; the accuracy will not be good enough. If so, then we divide the integration interval into two equal parts of size $\frac{1}{2}(b-a)$ each, and we repeat the process above in each part separately, calculating estimates I_1 and I_2 using one and two slices respectively, estimating the error, and checking to see if it is less than $h_2\delta$ (with the new value of h_2 now).

We keep on repeating this process, dividing each slice in half and in half again, as many times as necessary to achieve the desired accuracy in every slice. Different slices may be divided different numbers of times, and hence we may end up with different sized slices in different parts of the integration domain. The method automatically uses whatever size and number of slices is appropriate in each region.

a) Write a program using this method to calculate the integral

$$I = \int_0^{10} \frac{\sin^2 x}{x^2}\,\mathrm{d}x,$$

to an accuracy of $\epsilon = 10^{-4}$. Start by writing a function to calculate the integrand $f(x) = (\sin^2 x)/x^2$. Note that the limiting value of the integrand at $x = 0$ is 1. You'll probably have to include this point as a special case in your function using an if statement.

The best way to perform the integration itself is to make use of the technique of recursion, the ability of a Python function to call itself. (If you're not familiar with recursion, you may like to look at Exercise 2.13 on page 83 before doing this exercise.) Write a function `step(x1,x2,f1,f2)` that takes as arguments the beginning and end points x_1, x_2 of a slice and the values $f(x_1), f(x_2)$ of the integrand at those two points, and returns the value of the integral from x_1 to x_2. This function should evaluate the two estimates I_1 and I_2 of the integral from x_1 to x_2, calculated with one and two slices respectively, and the error $\frac{1}{3}(I_2 - I_1)$. If this error meets the target value, which is $\frac{1}{2}(x_2 - x_1)\delta$, then the calculation is complete and the function simply returns the value I_2. If the error fails to meet the target, then the function calls itself, twice, to evaluate the integral separately on the first and second halves of the interval and returns the sum of the two results. (And then *those* functions can call themselves, and so forth, subdividing the integral as many times as necessary to reach the required accuracy.)

Hint: As icing on the cake, when the error target is met and the function returns a value for the integral in the current slice, it can, in fact, return a slightly better value than the estimate I_2. Since you will already have calculated the value of the integrand $f(x)$ at x_1, x_2, and the midpoint $x_m = \frac{1}{2}(x_1 + x_2)$ in order to evaluate I_2, you can use those results to compute the improved Simpson's rule estimate, Eq. (5.7), for this slice. You just return the value $\frac{1}{6}h[f(x_1) + 4f(x_m) + f(x_2)]$ instead of the trapezoidal rule estimate $\frac{1}{4}h[f(x_1) + 2f(x_m) + f(x_2)]$ (where $h = x_2 - x_1$). This involves very little extra work, but gives a value that is more accurate by two orders in h. (Technically, this is an example of the method of "local extrapolation," although it's perhaps not obvious what we're extrapolating in this case. We'll discuss local extrapolation again when we study adaptive methods for the solution of differential equations in Section 8.4.)

b) Why does the function `step(x1,x2,f1,f2)` take not only the positions x_1 and x_2 as arguments, but also the values $f(x_1)$ and $f(x_2)$? Since we know the function $f(x)$, we could just calculate these values from x_1 and x_2. Nonetheless, it is a smart move to include the values of $f(x_1)$ and $f(x_2)$ as arguments to the function. Why?

c) Modify your program to make a plot of the integrand with dots added showing where the ends of each integration slice lie. You should see larger slices in portions of the integrand that follow reasonably straight lines (because the trapezoidal rule gives an accurate value for straight-line integrands) and smaller slices in portions with more curvature.

Exercise 5.21: Electric field of a charge distribution

Suppose we have a distribution of charges and we want to calculate the resulting electric field. One way to do this is to first calculate the electric potential ϕ and then take its gradient. For a point charge q at the origin, the electric potential at a distance r from the origin is $\phi = q/4\pi\epsilon_0 r$ and the electric field is $\mathbf{E} = -\nabla\phi$.

a) You have two charges, of $\pm 1\,\mathrm{C}$, $10\,\mathrm{cm}$ apart. Calculate the resulting electric potential on a $1\,\mathrm{m} \times 1\,\mathrm{m}$ square plane surrounding the charges and passing through them. Calculate the potential at $1\,\mathrm{cm}$ spaced points in a grid and make a visualization on the screen of the potential using a density plot.

b) Now calculate the partial derivatives of the potential with respect to x and y and hence find the electric field in the xy plane. Make a visualization of the field also. This is a little trickier than visualizing the potential, because the electric field has both magnitude and direction. One way to do it might be to make two density plots, one for the magnitude, and one for the direction, the latter using the "`hsv`" color scheme in `pylab`, which is a rainbow scheme that passes through all the colors but starts and ends with the same shade of red, which makes it suitable for representing things like directions or angles that go around the full circle and end up where they started. A more sophisticated visualization might use the arrow object from the `visual` package, drawing a grid of arrows with direction and length chosen to represent the field.

c) Now suppose you have a continuous distribution of charge over an $L \times L$ square. The charge density in Cm^{-2} is

$$\sigma(x,y) = q_0 \sin\frac{2\pi x}{L} \sin\frac{2\pi y}{L}.$$

Calculate and visualize the resulting electric field at $1\,\mathrm{cm}$-spaced points in 1 square meter of the xy plane for the case where $L = 10\,\mathrm{cm}$, the charge distribution is centered in the middle of the visualized area, and $q_0 = 100\,\mathrm{Cm}^{-2}$. You will have to perform a double integral over x and y, then differentiate the potential with respect to position to get the electric field. Choose whatever integration method seems appropriate for the integrals.

Exercise 5.22: Differentiating by integrating

If you are familiar with the calculus of complex variables, you may find the following technique useful and interesting.

Suppose we have a function $f(z)$ whose value we know not only on the real line but also for complex values of its argument. Then we can calculate derivatives of that function at any point z_0 by performing a contour integral, using the Cauchy derivative formula:

$$\left(\frac{\mathrm{d}^m f}{\mathrm{d}z^m}\right)_{z=z_0} = \frac{m!}{2\pi\mathrm{i}} \oint \frac{f(z)}{(z-z_0)^{m+1}}\,\mathrm{d}z,$$

where the integral is performed counterclockwise around any contour in the complex plane that surrounds the point z_0 but contains no poles in $f(z)$. Since numerical integration is significantly easier and more accurate than numerical differentiation, this formula provides us with a method for calculating derivatives—and especially multiple derivatives—accurately by turning them into integrals.

Suppose, for example, that we want to calculate derivatives of $f(z)$ at $z = 0$. Let us apply the Cauchy formula above using the trapezoidal rule to calculate the integral along a circular contour centered on the origin with radius 1. The trapezoidal rule will be slightly different from the version we are used to because the value of the interval h is now a complex number, and moreover is not constant from one slice of the integral to the next—it stays constant in modulus, but its argument changes from one slice to another.

We will divide our contour integral into N slices with sample points z_k distributed uniformly around the circular contour at the positions $z_k = \mathrm{e}^{\mathrm{i}2\pi k/N}$ for $k = 0\ldots N$. Then the distance between consecutive sample points is

$$h_k = z_{k+1} - z_k = \mathrm{e}^{\mathrm{i}2\pi(k+1)/N} - \mathrm{e}^{\mathrm{i}2\pi k/N},$$

and, introducing the shorthand $g(z) = f(z)/z^{m+1}$ for the integrand, the trapezoidal rule approximation to the integral is

$$\begin{aligned}\oint g(z)\,\mathrm{d}z &\simeq \sum_{k=0}^{N-1} \tfrac{1}{2}\bigl[g(z_{k+1}) + g(z_k)\bigr]\bigl[\mathrm{e}^{\mathrm{i}2\pi(k+1)/N} - \mathrm{e}^{\mathrm{i}2\pi k/N}\bigr] \\ &= \tfrac{1}{2}\biggl[\sum_{k=0}^{N-1} g(z_{k+1})\,\mathrm{e}^{\mathrm{i}2\pi(k+1)/N} - \sum_{k=0}^{N-1} g(z_k)\,\mathrm{e}^{\mathrm{i}2\pi k/N} \\ &\qquad - \sum_{k=0}^{N-1} g(z_{k+1})\,\mathrm{e}^{\mathrm{i}2\pi k/N} + \sum_{k=0}^{N-1} g(z_k)\,\mathrm{e}^{\mathrm{i}2\pi(k+1)/N}\biggr].\end{aligned}$$

Noting that $z_N = z_0$, the first two sums inside the brackets cancel each other in their entirety, and the remaining two sums are equal except for trivial phase factors, so the entire expression simplifies to

$$\begin{aligned}\oint g(z)\,\mathrm{d}z &\simeq \tfrac{1}{2}\bigl[\mathrm{e}^{\mathrm{i}2\pi/N} - \mathrm{e}^{-\mathrm{i}2\pi/N}\bigr]\sum_{k=0}^{N-1} g(z_k)\,\mathrm{e}^{\mathrm{i}2\pi k/N} \\ &\simeq \frac{2\pi\mathrm{i}}{N}\sum_{k=0}^{N-1} f(z_k)\,\mathrm{e}^{-\mathrm{i}2\pi km/N},\end{aligned}$$

where we have used the definition of $g(z)$ again. Combining this result with the Cauchy formula, we then have

$$\left(\frac{\mathrm{d}^m f}{\mathrm{d}z^m}\right)_{z=0} \simeq \frac{m!}{N}\sum_{k=0}^{N-1} f(z_k)\,\mathrm{e}^{-\mathrm{i}2\pi km/N}.$$

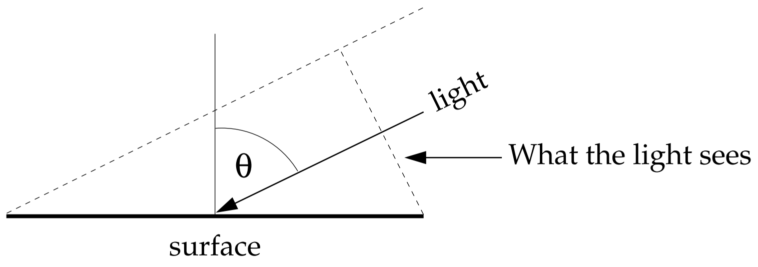
Write a program to calculate the first twenty derivatives of $f(z) = \mathrm{e}^{2z}$ at $z = 0$ using this formula with $N = 10000$. You will need to use the version of the `exp` function from the `cmath` package, which can handle complex arguments. You may also find the function `factorial` from the `math` package useful; it calculates factorials of integer arguments.

The correct value for the mth derivative in this case is easily shown to be 2^m, so it should be straightforward to tell if your program is working—the results should be powers of two, 2, 4, 8, 16, 32, etc. You should find that it is possible to get reasonably accurate results for all twenty derivatives rapidly using this technique. If you use standard difference formulas for the derivatives, on the other hand, you will find that you can calculate only the first three or four derivatives accurately before the numerical errors become so large that the results are useless. In this case, therefore, the Cauchy formula gives the better results.

The sum $\sum_k f(z_k)\,\mathrm{e}^{\mathrm{i}2\pi km/N}$ that appears in the formula above is known as the *discrete Fourier transform* of the complex samples $f(z_k)$. There exists an elegant technique for evaluating the Fourier transform for many values of m simultaneously, known as the *fast Fourier transform*, which could be useful in cases where the direct evaluation of the formula is slow. We will study the fast Fourier transform in detail in Chapter 7.

Exercise 5.23: Image processing and the STM

When light strikes a surface, the amount falling per unit area depends not only on the intensity of the light, but also on the angle of incidence. If the light makes an angle θ to the normal, it only "sees" $\cos\theta$ of area per unit of actual area on the surface:



So the intensity of illumination is $a\cos\theta$, if a is the raw intensity of the light. This simple physical law is a central element of 3D computer graphics. It allows us to calculate how light falls on three-dimensional objects and hence how they will look when illuminated from various angles.

Suppose, for instance, that we are looking down on the Earth from above and we see mountains. We know the height of the mountains $w(x,y)$ as a function of position in the plane, so the equation for the Earth's surface is simply $z = w(x,y)$, or equivalently $w(x,y) - z = 0$, and the normal vector $\mathbf{v}$ to the surface is given by the gradient of $w(x,y) - z$ thus:

$$\mathbf{v} = \nabla[w(x,y) - z] = \begin{pmatrix} \partial/\partial x \\ \partial/\partial y \\ \partial/\partial z \end{pmatrix} [w(x,y) - z] = \begin{pmatrix} \partial w/\partial x \\ \partial w/\partial y \\ -1 \end{pmatrix}.$$

Now suppose we have light coming in represented by a vector $\mathbf{a}$ with magnitude equal to the intensity of the light. Then the dot product of the vectors $\mathbf{a}$ and $\mathbf{v}$ is

$$\mathbf{a}\cdot\mathbf{v} = |\mathbf{a}|\,|\mathbf{v}|\cos\theta,$$

where θ is the angle between the vectors. Thus the intensity of illumination of the surface of the mountains is

$$I = |\mathbf{a}|\cos\theta = \frac{\mathbf{a}\cdot\mathbf{v}}{|\mathbf{v}|} = \frac{a_x(\partial w/\partial x) + a_y(\partial w/\partial y) - a_z}{\sqrt{(\partial w/\partial x)^2 + (\partial w/\partial y)^2 + 1}}.$$

Let's take a simple case where the light is shining horizontally with unit intensity, along a line an angle ϕ counter-clockwise from the east-west axis, so that $\mathbf{a} = (\cos\phi, \sin\phi, 0)$. Then our intensity of illumination simplifies to

$$I = \frac{\cos\phi\,(\partial w/\partial x) + \sin\phi\,(\partial w/\partial y)}{\sqrt{(\partial w/\partial x)^2 + (\partial w/\partial y)^2 + 1}}.$$

If we can calculate the derivatives of the height $w(x,y)$ and we know ϕ we can calculate the intensity at any point.

a) In the on-line resources you'll find a file called `altitude.txt`, which contains the altitude $w(x,y)$ in meters above sea level (or depth below sea level) of the surface of the Earth, measured on a grid of points (x,y). Write a program that reads this file and stores the data in an array. Then calculate the derivatives $\partial w/\partial x$ and $\partial w/\partial y$ at each grid point. Explain what method you used to calculate them and why. (Hint: You'll probably have to use more than one method to get every grid point, because awkward things happen at the edges of the grid.) To calculate the derivatives you'll need to know the value of h, the distance in meters between grid points, which is about 30 000 m in this case. (It's actually not precisely constant because we are representing the spherical Earth on a flat map, but $h = 30\,000$ m will give reasonable results.)

b) Now, using your values for the derivatives, calculate the intensity for each grid point, with $\phi = 45°$, and make a density plot of the resulting values in which the brightness of each dot depends on the corresponding intensity value. If you get it working right, the plot should look like a relief map of the world—you should be able to see the continents and mountain ranges in 3D. (Common problems include a map that is upside-down or sideways, or a relief map that is "inside-out," meaning the high regions look low and *vice versa*. Work with the details of your program until you get a map that looks right to you.)

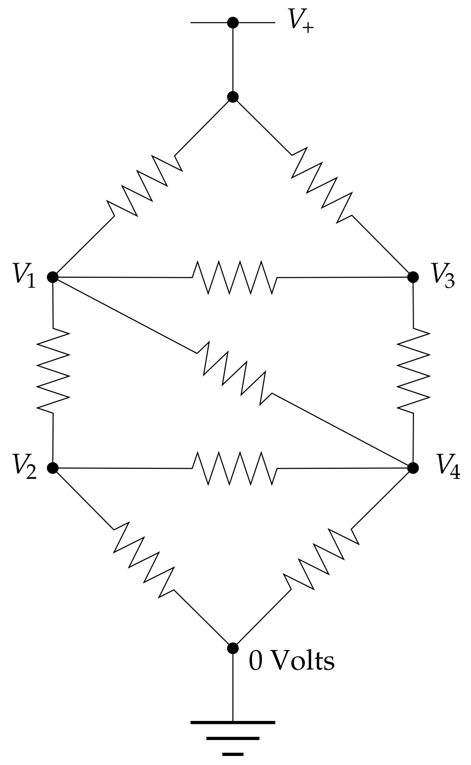
c) There is another file in the on-line resources called `stm.txt`, which contains a grid of values from scanning tunneling microscope measurements of the (111) surface of silicon. A scanning tunneling microscope (STM) is a device that measures the shape of surfaces at the atomic level by tracking a 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 as a function of position and the data in the file `stm.txt` contain just such a grid of values. Modify the program you just wrote to visualize the STM data and hence create a 3D picture of what the silicon surface looks like. The value of h for the derivatives in this case is around $h = 2.5$ (in arbitrary units).

Computational Physics

Exercises for Chapter 6

Exercise 6.1: A circuit of resistors

Consider the following circuit of resistors:



All the resistors have the same resistance R. The power rail at the top is at voltage $V_+ = 5\,\text{V}$. What are the other four voltages, V_1 to V_4?

To answer this question we use Ohm's law and the Kirchhoff current law, which says that the total net current flow out of (or into) any junction in a circuit must be zero. Thus for the junction at voltage V_1, for instance, we have

$$\frac{V_1 - V_2}{R} + \frac{V_1 - V_3}{R} + \frac{V_1 - V_4}{R} + \frac{V_1 - V_+}{R} = 0,$$

or equivalently

$$4V_1 - V_2 - V_3 - V_4 = V_+.$$

a) Write similar equations for the other three junctions with unknown voltages.

b) Write a program to solve the four resulting equations using Gaussian elimination and hence find the four voltages (or you can modify a program you already have, such as the program `gausselim.py` in Example 6.1).

Exercise 6.2:

a) Modify the program `gausselim.py` in Example 6.1 to incorporate partial pivoting (or you can write your own program from scratch if you prefer). Run your program and demonstrate that it gives the same answers as the original program when applied to Eq. (6.1)

b) Modify the program to solve the equations in (6.17) and show that it can find the solution to these as well, even though Gaussian elimination without pivoting fails.

Exercise 6.3: LU decomposition

This exercise invites you to write your own program to solve simultaneous equations using the method of LU decomposition.

a) Starting, if you wish, with the program for Gaussian elimination in Example 6.1 on page 218, write a Python function that calculates the LU decomposition of a matrix. The calculation is same as that for Gaussian elimination, except that at each step of the calculation you need to extract the appropriate elements of the matrix and assemble them to form the lower diagonal matrix **L** of Eq. (6.32). Test your function by calculating the LU decomposition of the matrix from Eq. (6.2), then multiplying the **L** and **U** you get and verifying that you recover the original matrix once more.

b) Build on your LU decomposition function to create a complete program to solve Eq. (6.2) by performing a double backsubstitution as described in this section. Solve the same equations using the function `solve` from the `numpy` package and verify that you get the same answer either way.

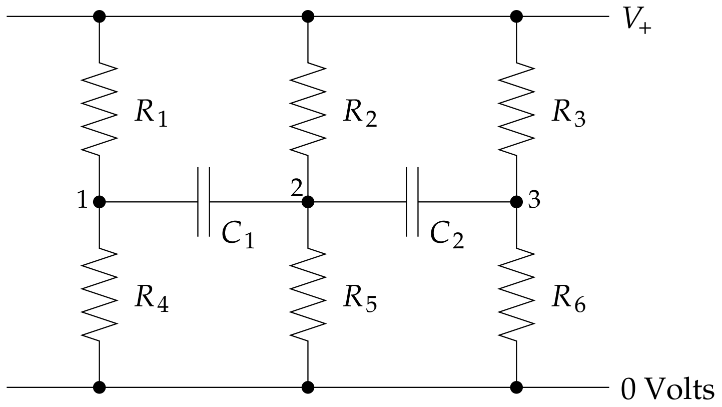
c) If you're feeling ambitious, try your hand at LU decomposition with partial pivoting. Partial pivoting works in the same way for LU decomposition as it does for Gaussian elimination, swapping rows to get the largest diagonal element as explained in Section 6.1.3, but the extension to LU decomposition requires two additional steps. First, every time you swap two rows you also have to swap the same rows in the matrix **L**. Second, when you use your LU decomposition to solve a set of equations $\mathbf{Ax} = \mathbf{v}$ you will also need to perform the same sequence of swaps on the vector **v** on the right-hand side. This means you need to record the swaps as you are doing the decomposition so that you can recreate them later. The simplest way to do this is to set up a list or array in which the value of the ith element records the row you swapped with on the ith step of the process. For instance, if you swapped the first row with the second then the second with the fourth, the first two elements of the list would be 2 and 4. Solving a set of equations for given **v** involves first performing the required sequence of swaps on the elements of **v** then performing a double backsubstitution as usual. (In ordinary Gaussian elimination with pivoting, one swaps the elements of **v** as the algorithm proceeds, rather than all at once, but the difference has no effect on the results, so it's fine to perform all the swaps at once if we wish.)

   Modify the function you wrote for part (a) to perform LU decomposition with partial pivoting. The function should return the matrices **L** and **U** for the LU decomposition of the swapped matrix, plus a list of the swaps made. Then modify the rest of your program to solve equations of the form $\mathbf{Ax} = \mathbf{v}$ using LU decomposition with pivoting. Test your program on the example from Eq. (6.17), which cannot be solved without pivoting because of the zero in the first element of the matrix. Check your results against a solution of the same equations using the `solve` function from `numpy`.

LU decomposition with partial pivoting is the most widely used method for the solution of simultaneous equations in practice. Precisely this method is used in the function `solve` from the `numpy` package. There's nothing wrong with using the `solve` function—it's well written, fast, and convenient. But it does nothing you haven't already done yourself if you've solved this exercise.

Exercise 6.4: Write a program to solve the resistor network problem of Exercise 6.1 on page 220 using the function `solve` from `numpy.linalg`. If you also did Exercise 6.1, you should check that you get the same answer both times.

Exercise 6.5: Here's a more complicated circuit problem:



The voltage V_+ is time-varying and sinusoidal of the form $V_+ = x_+ e^{i\omega t}$ with x_+ a constant. The resistors in the circuit can be treated using Ohm's law as usual. For the capacitors the charge Q and voltage V across them are related by the capacitor law $Q = CV$, where C is the capacitance. Differentiating both sides of this expression gives the current I flowing in on one side of the capacitor and out on the other:

$$I = \frac{dQ}{dt} = C\frac{dV}{dt}.$$

a) Assuming the voltages at the points labeled 1, 2, and 3 are of the form $V_1 = x_1 e^{i\omega t}$, $V_2 = x_2 e^{i\omega t}$, and $V_3 = x_3 e^{i\omega t}$, apply Kirchhoff's law at each of the three points, along with Ohm's law and the capacitor law, to show that the constants x_1, x_2, and x_3 satisfy the equations

$$\begin{aligned}\left(\frac{1}{R_1} + \frac{1}{R_4} + i\omega C_1\right)x_1 - i\omega C_1 x_2 &= \frac{x_+}{R_1},\\ -i\omega C_1 x_1 + \left(\frac{1}{R_2} + \frac{1}{R_5} + i\omega C_1 + i\omega C_2\right)x_2 - i\omega C_2 x_3 &= \frac{x_+}{R_2},\\ -i\omega C_2 x_2 + \left(\frac{1}{R_3} + \frac{1}{R_6} + i\omega C_2\right)x_3 &= \frac{x_+}{R_3}.\end{aligned}$$

b) Write a program to solve for x_1, x_2, and x_3 when

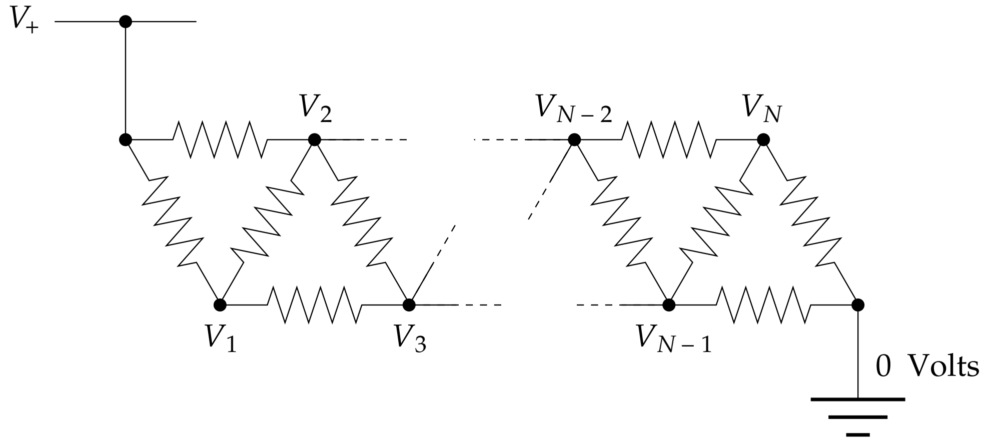
$$\begin{aligned} R_1 &= R_3 = R_5 = 1\,\text{k}\Omega, \\ R_2 &= R_4 = R_6 = 2\,\text{k}\Omega, \\ C_1 &= 1\,\mu\text{F}, \qquad C_2 = 0.5\,\mu\text{F}, \\ x_+ &= 3\,\text{V}, \qquad \omega = 1000\,\text{s}^{-1}. \end{aligned}$$

Notice that the matrix for this problem has complex elements. You will need to define a complex array to hold it, but you can still use the `solve` function just as before to solve the equations—it works with either real or complex arguments. Using your solution have your program calculate and print the amplitudes of the three voltages V_1, V_2, and V_3 and their phases in degrees. (Hint: You may find the functions `polar` or `phase` in the `cmath` package useful. If `z` is a complex number then "`r,theta = polar(z)`" will return the modulus and phase (in radians) of `z` and "`theta = phase(z)`" will return the phase alone.)

Exercise 6.6: Starting with either the program `springs.py` on page 237 or `springsb.py` on page 238, remove the code that makes a graph of the results and replace it with code that creates an animation of the masses as they vibrate back and forth, their displacements relative to their resting positions being given by the real part of Eq. (6.53). For clarity, assume that the resting positions are two units apart in a horizontal line. At a minimum your animation should show each of the individual masses, perhaps as small spheres. (Spheres of radius about 0.2 or 0.3 seem to work well.)

Exercise 6.7: A chain of resistors

Consider a long chain of resistors wired up like this:



All the resistors have the same resistance R. The power rail at the top is at voltage $V_+ = 5$V. The problem is to find the voltages $V_1 \ldots V_N$ at the internal points in the circuit.

a) Using Ohm's law and the Kirchhoff current law, which says that the total net current flow out of (or into) any junction in a circuit must be zero, show that the voltages $V_1 \dots V_N$ satisfy the equations

$$\begin{aligned} 3V_1 - V_2 - V_3 &= V_+, \\ -V_1 + 4V_2 - V_3 - V_4 &= V_+, \\ &\vdots \\ -V_{i-2} - V_{i-1} + 4V_i - V_{i+1} - V_{i+2} &= 0, \\ &\vdots \\ -V_{N-3} - V_{N-2} + 4V_{N-1} - V_N &= 0, \\ -V_{N-2} - V_{N-1} + 3V_N &= 0. \end{aligned}$$

Express these equations in vector form $\mathbf{Av} = \mathbf{w}$ and find the values of the matrix $\mathbf{A}$ and the vector $\mathbf{w}$.

b) Write a program to solve for the values of the V_i when there are $N = 6$ internal junctions with unknown voltages. (Hint: All the values of V_i should lie between zero and 5V. If they don't, something is wrong.)

c) Now repeat your calculation for the case where there are $N = 10\,000$ internal junctions. This part is not possible using standard tools like the `solve` function. You need to make use of the fact that the matrix $\mathbf{A}$ is banded and use the `banded` function from the file `banded.py`, discussed in Appendix E.

Exercise 6.8: The QR algorithm

In this exercise you'll write a program to calculate the eigenvalues and eigenvectors of a real symmetric matrix using the QR algorithm. The first challenge is to write a program that finds the QR decomposition of a matrix. Then we'll use that decomposition to find the eigenvalues.

As described above, the QR decomposition expresses a real square matrix $\mathbf{A}$ in the form $\mathbf{A} = \mathbf{QR}$, where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper-triangular matrix. Given an $N \times N$ matrix $\mathbf{A}$ we can compute the QR decomposition as follows.

Let us think of the matrix as a set of N column vectors $\mathbf{a}_0 \dots \mathbf{a}_{N-1}$ thus:

$$\mathbf{A} = \begin{pmatrix} | & | & | & \cdots \\ \mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2 & \cdots \\ | & | & | & \cdots \end{pmatrix},$$

where we have numbered the vectors in Python fashion, starting from zero, which will be convenient when writing the program. We now define two new sets of vectors $\mathbf{u}_0 \dots \mathbf{u}_{N-1}$ and $\mathbf{q}_0 \dots \mathbf{q}_{N-1}$ as follows:

$$\begin{aligned} \mathbf{u}_0 &= \mathbf{a}_0, & \mathbf{q}_0 &= \frac{\mathbf{u}_0}{|\mathbf{u}_0|}, \\ \mathbf{u}_1 &= \mathbf{a}_1 - (\mathbf{q}_0 \cdot \mathbf{a}_1)\mathbf{q}_0, & \mathbf{q}_1 &= \frac{\mathbf{u}_1}{|\mathbf{u}_1|}, \\ \mathbf{u}_2 &= \mathbf{a}_2 - (\mathbf{q}_0 \cdot \mathbf{a}_2)\mathbf{q}_0 - (\mathbf{q}_1 \cdot \mathbf{a}_2)\mathbf{q}_1, & \mathbf{q}_2 &= \frac{\mathbf{u}_2}{|\mathbf{u}_2|}, \end{aligned}$$

and so forth. The general formulas for calculating $\mathbf{u}_i$ and $\mathbf{q}_i$ are

$$\mathbf{u}_i = \mathbf{a}_i - \sum_{j=0}^{i-1} (\mathbf{q}_j \cdot \mathbf{a}_i)\mathbf{q}_j, \qquad \mathbf{q}_i = \frac{\mathbf{u}_i}{|\mathbf{u}_i|}.$$

a) Show, by induction or otherwise, that the vectors $\mathbf{q}_i$ are orthonormal, i.e., that they satisfy

$$\mathbf{q}_i \cdot \mathbf{q}_j = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

Now, rearranging the definitions of the vectors, we have

$$\begin{aligned} \mathbf{a}_0 &= |\mathbf{u}_0|\,\mathbf{q}_0, \\ \mathbf{a}_1 &= |\mathbf{u}_1|\,\mathbf{q}_1 + (\mathbf{q}_0 \cdot \mathbf{a}_1)\mathbf{q}_0, \\ \mathbf{a}_2 &= |\mathbf{u}_2|\,\mathbf{q}_2 + (\mathbf{q}_0 \cdot \mathbf{a}_2)\mathbf{q}_0 + (\mathbf{q}_1 \cdot \mathbf{a}_2)\mathbf{q}_1, \end{aligned}$$

and so on. Or we can group the vectors $\mathbf{q}_i$ together as the columns of a matrix and write all of these equations as a single matrix equation

$$\mathbf{A} = \begin{pmatrix} | & | & | & \cdots \\ \mathbf{a}_0 & \mathbf{a}_1 & \mathbf{a}_2 & \cdots \\ | & | & | & \cdots \end{pmatrix} = \begin{pmatrix} | & | & | & \cdots \\ \mathbf{q}_0 & \mathbf{q}_1 & \mathbf{q}_2 & \cdots \\ | & | & | & \cdots \end{pmatrix} \begin{pmatrix} |\mathbf{u}_0| & \mathbf{q}_0 \cdot \mathbf{a}_1 & \mathbf{q}_0 \cdot \mathbf{a}_2 & \cdots \\ 0 & |\mathbf{u}_1| & \mathbf{q}_1 \cdot \mathbf{a}_2 & \cdots \\ 0 & 0 & |\mathbf{u}_2| & \cdots \end{pmatrix}.$$

(If this looks complicated it's worth multiplying out the matrices on the right to verify for yourself that you get the correct expressions for the $\mathbf{a}_i$.)

Notice now that the first matrix on the right-hand side of this equation, the matrix with columns $\mathbf{q}_i$, is orthogonal, because the vectors $\mathbf{q}_i$ are orthonormal, and the second matrix is upper triangular. In other words, we have found the QR decomposition $\mathbf{A} = \mathbf{QR}$. The matrices $\mathbf{Q}$ and $\mathbf{R}$ are

$$\mathbf{Q} = \begin{pmatrix} | & | & | & \cdots \\ \mathbf{q}_0 & \mathbf{q}_1 & \mathbf{q}_2 & \cdots \\ | & | & | & \cdots \end{pmatrix}, \qquad \mathbf{R} = \begin{pmatrix} |\mathbf{u}_0| & \mathbf{q}_0 \cdot \mathbf{a}_1 & \mathbf{q}_0 \cdot \mathbf{a}_2 & \cdots \\ 0 & |\mathbf{u}_1| & \mathbf{q}_1 \cdot \mathbf{a}_2 & \cdots \\ 0 & 0 & |\mathbf{u}_2| & \cdots \end{pmatrix}.$$

b) Write a Python function that takes as its argument a real square matrix $\mathbf{A}$ and returns the two matrices $\mathbf{Q}$ and $\mathbf{R}$ that form its QR decomposition. As a test case, try out your function on the matrix

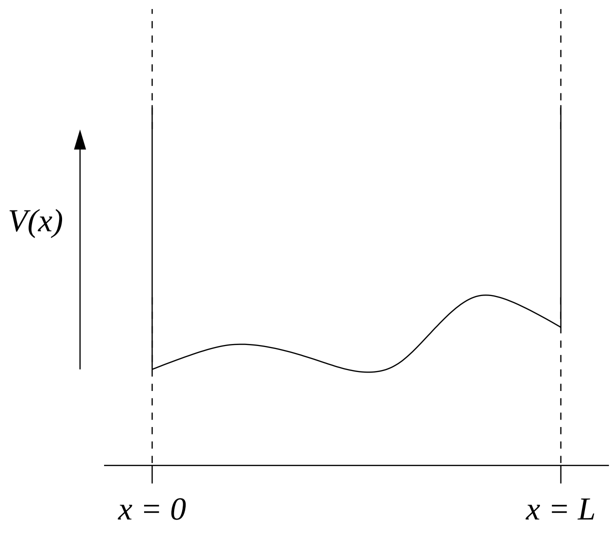
$$\mathbf{A} = \begin{pmatrix} 1 & 4 & 8 & 4 \\ 4 & 2 & 3 & 7 \\ 8 & 3 & 6 & 9 \\ 4 & 7 & 9 & 2 \end{pmatrix}.$$

Check the results by multiplying $\mathbf{Q}$ and $\mathbf{R}$ together to recover the original matrix $\mathbf{A}$ again.

c) Using your function, write a complete program to calculate the eigenvalues and eigenvectors of a real symmetric matrix using the QR algorithm. Continue the calculation until the magnitude of every off-diagonal element of the matrix is smaller than 10^{-6}. Test your program on the example matrix above. You should find that the eigenvalues are 1, 21, -3, and -8.

Exercise 6.9: Asymmetric quantum well

Quantum mechanics can be formulated as a matrix problem and solved on a computer using linear algebra methods. Suppose, for example, we have a particle of mass M in a one-dimensional quantum well of width L, but not a square well like the examples you've probably seen before. Suppose instead that the potential $V(x)$ varies somehow inside the well:



We cannot solve such problems analytically in general, but we can solve them on the computer.

In a pure state of energy E, the spatial part of the wavefunction obeys the time-independent Schrödinger equation $\hat{H}\psi(x) = E\psi(x)$, where the Hamiltonian operator $\hat{H}$ is given by

$$\hat{H} = -\frac{\hbar^2}{2M}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x).$$

For simplicity, let's assume that the walls of the well are infinitely high, so that the wavefunction is zero outside the well, which means it must *go to* zero at $x = 0$ and $x = L$. In that case, the wavefunction can be expressed as a Fourier sine series thus:

$$\psi(x) = \sum_{n=1}^{\infty} \psi_n \sin\frac{\pi n x}{L},$$

where $\psi_1, \psi_2, \ldots$ are the Fourier coefficients.

a) Noting that, for m, n positive integers

$$\int_0^L \sin\frac{\pi m x}{L}\sin\frac{\pi n x}{L}\,\mathrm{d}x = \begin{cases} L/2 & \text{if } m = n, \\ 0 & \text{otherwise,} \end{cases}$$

show that the Schrödinger equation $\hat{H}\psi = E\psi$ implies that

$$\sum_{n=1}^{\infty} \psi_n \int_0^L \sin\frac{\pi m x}{L}\hat{H}\sin\frac{\pi n x}{L}\,\mathrm{d}x = \tfrac{1}{2}LE\psi_m.$$

Hence, defining a matrix $\mathbf{H}$ with elements

$$\begin{aligned} H_{mn} &= \frac{2}{L}\int_0^L \sin\frac{\pi m x}{L}\hat{H}\sin\frac{\pi n x}{L}\,\mathrm{d}x \\ &= \frac{2}{L}\int_0^L \sin\frac{\pi m x}{L}\left[-\frac{\hbar^2}{2M}\frac{\mathrm{d}^2}{\mathrm{d}x^2}+V(x)\right]\sin\frac{\pi n x}{L}\,\mathrm{d}x, \end{aligned}$$

show that Schrödinger's equation can be written in matrix form as $\mathbf{H}\boldsymbol{\psi} = E\boldsymbol{\psi}$, where $\boldsymbol{\psi}$ is the vector $(\psi_1, \psi_2, \ldots)$. Thus $\boldsymbol{\psi}$ is an eigenvector of the *Hamiltonian matrix* $\mathbf{H}$ with eigenvalue E. If we can calculate the eigenvalues of this matrix, then we know the allowed energies of the particle in the well.

b) For the case $V(x) = ax/L$, evaluate the integral in H_{mn} analytically and so find a general expression for the matrix element H_{mn}. Show that the matrix is real and symmetric. You'll probably find it useful to know that

$$\int_0^L x\sin\frac{\pi m x}{L}\sin\frac{\pi n x}{L}\,\mathrm{d}x = \begin{cases} 0 & \text{if } m \neq n \text{ and both even or both odd,} \\ -\left(\dfrac{2L}{\pi}\right)^2 \dfrac{mn}{(m^2-n^2)^2} & \text{if } m \neq n \text{ and one is even, one is odd,} \\ L^2/4 & \text{if } m = n. \end{cases}$$

Write a Python program to evaluate your expression for H_{mn} for arbitrary m and n when the particle in the well is an electron, the well has width 5 Å, and $a = 10\,\mathrm{eV}$. (The mass and charge of an electron are 9.1094×10^{-31} kg and 1.6022×10^{-19} C respectively.)

c) The matrix $\mathbf{H}$ is in theory infinitely large, so we cannot calculate all its eigenvalues. But we can get a pretty accurate solution for the first few of them by cutting off the matrix after the first few elements. Modify the program you wrote for part (b) above to create a 10×10 array of the elements of $\mathbf{H}$ up to $m, n = 10$. Calculate the eigenvalues of this matrix using the appropriate function from `numpy.linalg` and hence print out, in units of electron volts, the first ten energy levels of the quantum well, within this approximation. You should find, for example, that the ground-state energy of the system is around 5.84 eV. (Hint: Bear in mind that matrix indices in Python start at zero, while the indices in standard algebraic expressions, like those above, start at one. You will need to make allowances for this in your program.)

d) Modify your program to use a 100×100 array instead and again calculate the first ten energy eigenvalues. Comparing with the values you calculated in part (c), what do you conclude about the accuracy of the calculation?

e) Now modify your program once more to calculate the wavefunction $\psi(x)$ for the ground state and the first two excited states of the well. Use your results to make a graph with three curves showing the probability density $|\psi(x)|^2$ as a function of x in each of these three states. Pay special attention to the normalization of the wavefunction—it should satisfy the condition $\int_0^L |\psi(x)|^2\,\mathrm{d}x = 1$. Is this true of your wavefunction?

Exercise 6.10: Consider the equation $x = 1 - \mathrm{e}^{-cx}$, where c is a known parameter and x is unknown. This equation arises in a variety of situations, including the physics of contact processes, mathematical models of epidemics, and the theory of random graphs.

a) Write a program to solve this equation for x using the relaxation method for the case $c = 2$. Calculate your solution to an accuracy of at least 10^{-6}.

b) Modify your program to calculate the solution for values of c from 0 to 3 in steps of 0.01 and make a plot of x as a function of c. You should see a clear transition from a regime in which $x = 0$ to a regime of nonzero x. This is another example of a phase transition. In physics this transition is known as the *percolation transition*; in epidemiology it is the *epidemic threshold*.

Exercise 6.11: Overrelaxation

If you did not already do Exercise 6.10, you should do it before this one.

The ordinary relaxation method involves iterating the equation $x' = f(x)$, starting from an initial guess, until it converges. As we have seen, this is often a fast and easy way to find solutions to nonlinear equations. However, it is possible in some cases to make the method work even faster using the technique of *overrelaxation*. Suppose our initial guess at the solution of a particular equation is, say, $x = 1$, and the final, true solution is $x = 5$. After the first step of the iterative process, we might then see a value of, say, $x = 3$. In the overrelaxation method, we observe this value and note that x is increasing, then we deliberately overshoot the calculated value, in the hope that this will get us closer to the final solution—in this case we might pass over $x = 3$ and go straight to a value of $x = 4$ perhaps, which is closer to the final solution of $x = 5$ and hence should get us to that solution quicker. The overrelaxation method provides a formula for performing this kind of overshooting in a controlled fashion and often, though not always, it does get us to our solution faster. In detail, it works as follows.

We can rewrite the equation $x' = f(x)$ in the form $x' = x + \Delta x$, where

$$\Delta x = x' - x = f(x) - x.$$

The overrelaxation method involves iteration of the modified equation

$$x' = x + (1+\omega)\,\Delta x,$$

(keeping the definition of Δx the same). If the parameter ω is zero, then this is the same as the ordinary relaxation method, but for $\omega > 0$ the method takes the amount Δx by which the value of x would have been changed and changes by a little more. Using $\Delta x = f(x) - x$, we can also write x' as

$$x' = x + (1+\omega)\big[f(x) - x\big] = (1+\omega)f(x) - \omega x,$$

which is the form in which it is usually written.

For the method to work the value of ω must be chosen correctly, although there is some wiggle room—there is an optimal value, but other values close to it will typically also give good results. Unfortunately, there is no general theory that tells us what the optimal value is. Usually it is found by trial and error.

a) Derive an equivalent of Eq. (6.81) for the overrelaxation method and hence show that the error on x', the equivalent of Eq. (6.83), is given by

$$\epsilon' \simeq \frac{x - x'}{1 - 1/[(1+\omega)f'(x) - \omega]}.$$

b) Consider again the equation $x = 1 - e^{-cx}$ that we solved in Exercise 6.10. Take the program you wrote for part (a) of that exercise, which solved the equation for the case $c = 2$, and modify it to print out the number of iterations it takes to converge to a solution accurate to 10^{-6}.

c) Now write a new program (or modify the previous one) to solve the same equation $x = 1 - e^{-cx}$ for $c = 2$, again to an accuracy of 10^{-6}, but this time using overrelaxation. Have your program print out the answers it finds along with the number of iterations it took to find them. Experiment with different values of ω to see how fast you can get the method to converge. A value of $\omega = 0.5$ is a reasonable starting point. With some trial and error you should be able to get the calculation to converge about twice as fast as the simple relaxation method, i.e., in about half as many iterations.

d) Are there any circumstances under which using a value $\omega < 0$ would help us find a solution faster than we can with the ordinary relaxation method? (Hint: The answer is yes, but why?)

Exercise 6.12: The biochemical process of *glycolysis*, the breakdown of glucose in the body to release energy, can be modeled by the equations

$$\frac{dx}{dt} = -x + ay + x^2y, \qquad \frac{dy}{dt} = b - ay - x^2y.$$

Here x and y represent concentrations of two chemicals, ADP and F6P, and a and b are positive constants. One of the important features of nonlinear linear equations like these is their *stationary points*, meaning values of x and y at which the derivatives of both variables become zero simultaneously, so that the variables stop changing and become constant in time. Setting the derivatives to zero above, the stationary points of our glycolysis equations are solutions of

$$-x + ay + x^2y = 0, \qquad b - ay - x^2y = 0.$$

a) Demonstrate analytically that the solution of these equations is

$$x = b, \qquad y = \frac{b}{a + b^2}.$$

b) Show that the equations can be rearranged to read

$$x = y(a + x^2), \qquad y = \frac{b}{a + x^2}$$

and write a program to solve these for the stationary point using the relaxation method with $a = 1$ and $b = 2$. You should find that the method fails to converge to a solution in this case.

c) Find a different way to rearrange the equations such that when you apply the relaxation method again it now converges to a fixed point and gives a solution. Verify that the solution you get agrees with part (a).

Exercise 6.13: Wien's displacement constant

Planck's radiation law tells us that the intensity of radiation per unit area and per unit wavelength λ from a black body at temperature T is

$$I(\lambda) = \frac{2\pi hc^2\lambda^{-5}}{e^{hc/\lambda k_B T} - 1},$$

where h is Planck's constant, c is the speed of light, and k_B is Boltzmann's constant.

a) Show by differentiating that the wavelength λ at which the emitted radiation is strongest is the solution of the equation

$$5e^{-hc/\lambda k_B T} + \frac{hc}{\lambda k_B T} - 5 = 0.$$

Make the substitution $x = hc/\lambda k_B T$ and hence show that the wavelength of maximum radiation obeys the *Wien displacement law*:

$$\lambda = \frac{b}{T},$$

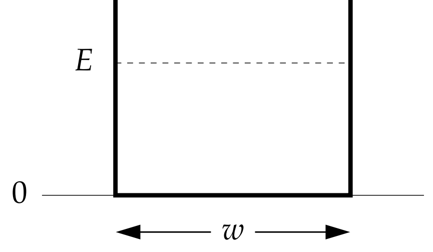
where the so-called *Wien displacement constant* is $b = hc/k_B x$, and x is the solution to the nonlinear equation

$$5e^{-x} + x - 5 = 0.$$

b) Write a program to solve this equation to an accuracy of $\epsilon = 10^{-6}$ using the binary search method, and hence find a value for the displacement constant.

c) The displacement law is the basis for the method of *optical pyrometry*, a method for measuring the temperatures of objects by observing the color of the thermal radiation they emit. The method is commonly used to estimate the surface temperatures of astronomical bodies, such as the Sun. The wavelength peak in the Sun's emitted radiation falls at $\lambda = 502\,\text{nm}$. From the equations above and your value of the displacement constant, estimate the surface temperature of the Sun.

Exercise 6.14: Consider a square potential well of width w, with walls of height V:



Using Schrödinger's equation, it can be shown that the allowed energies E of a single quantum particle of mass m trapped in the well are solutions of

$$\tan\sqrt{w^2 mE/2\hbar^2} = \begin{cases} \sqrt{(V-E)/E} & \text{for the even numbered states,} \\ -\sqrt{E/(V-E)} & \text{for the odd numbered states,} \end{cases}$$

where the states are numbered starting from 0, with the ground state being state 0, the first excited state being state 1, and so forth.

a) For an electron (mass 9.1094×10^{-31} kg) in a well with $V = 20\,\text{eV}$ and $w = 1\,\text{nm}$, write a Python program to plot the three quantities

$$y_1 = \tan\sqrt{w^2 mE/2\hbar^2}, \qquad y_2 = \sqrt{\frac{V-E}{E}}, \qquad y_3 = -\sqrt{\frac{E}{V-E}},$$

on the same graph, as a function of E from $E = 0$ to $E = 20\,\text{eV}$. From your plot make approximate estimates of the energies of the first six energy levels of the particle.

b) Write a second program to calculate the values of the first six energy levels in electron volts to an accuracy of 0.001 eV using binary search.

Exercise 6.15: The roots of a polynomial

Consider the sixth-order polynomial

$$P(x) = 924x^6 - 2772x^5 + 3150x^4 - 1680x^3 + 420x^2 - 42x + 1.$$

There is no general formula for the roots of a sixth-order polynomial, but one can find them easily enough using a computer.

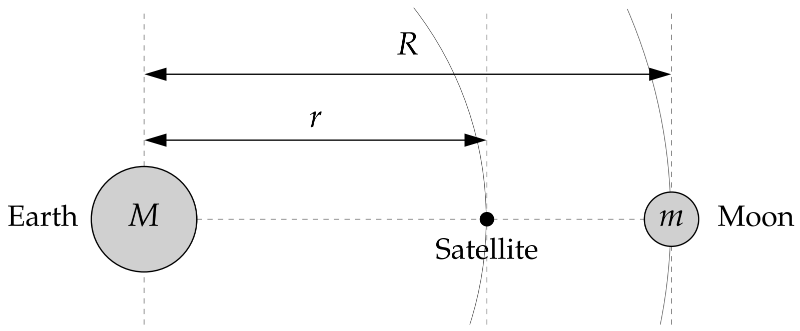
a) Make a plot of $P(x)$ from $x = 0$ to $x = 1$ and by inspecting it find rough values for the six roots of the polynomial—the points at which the function is zero.

b) Write a Python program to solve for the positions of all six roots to at least ten decimal places of accuracy, using Newton's method.

Note that the polynomial in this example is just the sixth Legendre polynomial (mapped onto the interval from zero to one), so the calculation performed here is the same as finding the integration points for 6-point Gaussian quadrature (see Section 5.6.2), and indeed Newton's method is the method of choice for calculating Gaussian quadrature points.

Exercise 6.16: The Lagrange point

There is a magical point between the Earth and the Moon, called the L_1 Lagrange point, at which a satellite will orbit the Earth in perfect synchrony with the Moon, staying always in between the two. This works because the inward pull of the Earth and the outward pull of the Moon combine to create exactly the needed centripetal force that keeps the satellite in its orbit. Here's the setup:



a) Assuming circular orbits, and assuming that the Earth is much more massive than either the Moon or the satellite, show that the distance r from the center of the Earth to the L_1 point satisfies

$$\frac{GM}{r^2} - \frac{Gm}{(R-r)^2} = \omega^2 r,$$

where M and m are the Earth and Moon masses, G is Newton's gravitational constant, and ω is the angular velocity of both the Moon and the satellite.

b) The equation above is a fifth-order polynomial equation in r (also called a quintic equation). Such equations cannot be solved exactly in closed form, but it's straightforward to solve them numerically. Write a program that uses either Newton's method or the secant method to solve for the distance r from the Earth to the L_1 point. Compute a solution accurate to at least four significant figures.

The values of the various parameters are:

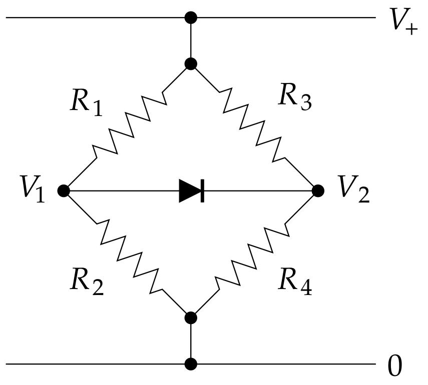
$$\begin{aligned} G &= 6.674 \times 10^{-11}\,\mathrm{m^3 kg^{-1} s^{-2}}, \\ M &= 5.974 \times 10^{24}\,\mathrm{kg}, \\ m &= 7.348 \times 10^{22}\,\mathrm{kg}, \\ R &= 3.844 \times 10^{8}\,\mathrm{m}, \\ \omega &= 2.662 \times 10^{-6}\,\mathrm{s^{-1}}. \end{aligned}$$

You will also need to choose a suitable starting value for r, or two starting values if you use the secant method.

Exercise 6.17: Nonlinear circuits

Exercise 6.1 used regular simultaneous equations to solve for the behavior of circuits of resistors. Resistors are linear—current is proportional to voltage—and the resulting equations we need to solve are therefore also linear and can be solved by standard matrix methods. Real circuits, however, often include nonlinear components. To solve for the behavior of these circuits we need to solve nonlinear equations.

Consider the following simple circuit, a variation on the classic Wheatstone bridge:



The resistors obey the normal Ohm law, but the diode obeys the diode equation:

$$I = I_0(e^{V/V_T} - 1),$$

where V is the voltage across the diode and I_0 and V_T are constants.

a) The Kirchhoff current law says that the total net current flowing into or out of every point in a circuit must be zero. Applying the law to voltage V_1 in the circuit above we get

$$\frac{V_1 - V_+}{R_1} + \frac{V_1}{R_2} + I_0\left[e^{(V_1-V_2)/V_T} - 1\right] = 0.$$

Derive the corresponding equation for voltage V_2.

b) Solve the two nonlinear equations for the voltages V_1 and V_2 with the conditions

$$\begin{aligned} V_+ &= 5\,\mathrm{V}, \\ R_1 &= 1\,\mathrm{k\Omega}, & R_2 &= 4\,\mathrm{k\Omega}, & R_3 &= 3\,\mathrm{k\Omega}, & R_4 &= 2\,\mathrm{k\Omega}, \\ I_0 &= 3\,\mathrm{nA}, & V_T &= 0.05\,\mathrm{V}. \end{aligned}$$

You can use either the relaxation method or Newton's method to solve the equations. If you use Newton's method you can solve Eq. (6.108) for $\Delta\mathbf{x}$ using the function `solve()` from `numpy.linalg` if you want to, but in this case the matrix is only a 2×2 matrix, so it's easy to calculate the inverse directly too.

c) The electronic engineer's rule of thumb for diodes is that the voltage across a (forward biased) diode is always about 0.6 volts. Confirm that your results agree with this rule.

Exercise 6.18: The temperature of a light bulb

An incandescent light bulb is a simple device—it contains a filament, usually made of tungsten, heated by the flow of electricity until it becomes hot enough to radiate thermally. Essentially all of the power consumed by such a bulb is radiated as electromagnetic energy, but some of the radiation is not in the visible wavelengths, which means it is useless for lighting purposes.

Let us define the efficiency of a light bulb to be the fraction of the radiated energy that falls in the visible band. It's a good approximation to assume that the radiation from a filament at temperature T obeys the Planck radiation law, meaning that the power radiated per unit wavelength λ obeys

$$I(\lambda) = 2\pi A h c^2 \frac{\lambda^{-5}}{e^{hc/\lambda k_B T} - 1},$$

where A is the surface area of the filament, h is Planck's constant, c is the speed of light, and k_B is Boltzmann's constant. The visible wavelengths run from $\lambda_1 = 390\,\text{nm}$ to $\lambda_2 = 750\,\text{nm}$, so the total energy radiated in the visible window is $\int_{\lambda_1}^{\lambda_2} I(\lambda)\,d\lambda$ and the total energy at all wavelengths is $\int_0^\infty I(\lambda)\,d\lambda$. Dividing one expression by the other and substituting for $I(\lambda)$ from above, we get an expression for the efficiency η of the light bulb thus:

$$\eta = \frac{\int_{\lambda_1}^{\lambda_2} \lambda^{-5}/(e^{hc/\lambda k_B T} - 1)\,d\lambda}{\int_0^\infty \lambda^{-5}/(e^{hc/\lambda k_B T} - 1)\,d\lambda},$$

where the leading constants and the area A have canceled out. Making the substitution $x = hc/\lambda k_B T$, this can also be written as

$$\eta = \frac{\int_{hc/\lambda_2 k_B T}^{hc/\lambda_1 k_B T} x^3/(e^x - 1)\,dx}{\int_0^\infty x^3/(e^x - 1)\,dx} = \frac{15}{\pi^4} \int_{hc/\lambda_2 k_B T}^{hc/\lambda_1 k_B T} \frac{x^3}{e^x - 1}\,dx,$$

where we have made use of the known exact value of the integral in the denominator.

a) Write a Python function that takes a temperature T as its argument and calculates the value of η for that temperature from the formula above. The integral in the formula cannot be done analytically, but you can do it numerically using any method of your choice. (For instance, Gaussian quadrature with 100 sample points works fine.) Use your function to make a graph of η as a function of temperature between 300 K and 10 000 K. You should see that there is an intermediate temperature where the efficiency is a maximum.

b) Calculate the temperature of maximum efficiency of the light bulb to within 1 K using golden ratio search. (Hint: An accuracy of 1 K is the equivalent of a few parts in ten thousand in this case. To get this kind of accuracy in your calculation you'll need to use values for the fundamental constants that are suitably accurate, i.e., you will need values accurate to several significant figures.)

c) Is it practical to run a tungsten-filament light bulb at the temperature you found? If not, why not?

COMPUTATIONAL PHYSICS

EXERCISES FOR CHAPTER 7

Exercise 7.1: Fourier transforms of simple functions

Write Python programs to calculate the coefficients in the discrete Fourier transforms of the following periodic functions sampled at $N = 1000$ evenly spaced points, and make plots of their amplitudes similar to the plot shown in Fig. 7.4:

a) A single cycle of a square-wave with amplitude 1
b) The sawtooth wave $y_n = n$
c) The modulated sine wave $y_n = \sin(\pi n/N)\sin(20\pi n/N)$

If you wish you can use the Fourier transform function from the file `dft.py` as a starting point for your program.

Exercise 7.2: Detecting periodicity

In the on-line resources there is 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 representing the month and the second being the sunspot number.

a) Write a program that reads the data in the file and makes a graph of sunspots as a function of time. You should see that the number of sunspots has fluctuated on a regular cycle for as long as observations have been recorded. Make an estimate of the length of the cycle in months.
b) Modify your program to calculate the Fourier transform of the sunspot data and then make a graph of the magnitude squared $|c_k|^2$ of the Fourier coefficients as a function of k—also called the *power spectrum* of the sunspot signal. You should see that there is a noticeable peak in the power spectrum at a nonzero value of k. The appearance of this peak tells us that there is one frequency in the Fourier series that has a higher amplitude than the others around it—meaning that there is a large sine-wave term with this frequency, which corresponds to the periodic wave you can see in the original data.
c) Find the approximate value of k to which the peak corresponds. What is the period of the sine wave with this value of k? You should find that the period corresponds roughly to the length of the cycle that you estimated in part (a).

This kind of Fourier analysis is a sensitive method for detecting periodicity in signals. Even in cases where it is not clear to the eye that there is a periodic component to a signal, it may still be possible to find one using a Fourier transform.

Exercise 7.3: Fourier transforms of musical instruments

In the on-line resources you will find files called `piano.txt` and `trumpet.txt`, which contain data representing the waveform of a single note, played on, respectively, a piano and a trumpet.

a) Write a program that loads a waveform from one of these files, plots it, then calculates its discrete Fourier transform and plots the magnitudes of the first 10 000 coefficients in a manner similar to Fig. 7.4. Note that you will have to use a fast Fourier transform for the calculation because there are too many samples in the files to do the transforms the slow way in any reasonable amount of time.

   Apply your program to the piano and trumpet waveforms and discuss briefly what one can conclude about the sound of the piano and trumpet from the plots of Fourier coefficients.

b) Both waveforms were recorded at the industry-standard rate of 44 100 samples per second and both instruments were playing the same musical note when the recordings were made. From your Fourier transform results calculate what note they were playing. (Hint: The musical note middle C has a frequency of 261 Hz.)

Exercise 7.4: Fourier filtering and smoothing

In the on-line resources you'll find a file called `dow.txt`. It contains the daily closing value for each business day from late 2006 until the end of 2010 of the Dow Jones Industrial Average, which is a measure of average prices on the US stock market.

Write a program to do the following:

a) Read in the data from `dow.txt` and plot them on a graph.
b) Calculate the coefficients of the discrete Fourier transform of the data using the function `rfft` from `numpy.fft`, which produces an array of $\frac{1}{2}N + 1$ complex numbers.
c) Now set all but the first 10% of the elements of this array to zero (i.e., set the last 90% to zero but keep the values of the first 10%).
d) Calculate the inverse Fourier transform of the resulting array, zeros and all, using the function `irfft`, and plot it on the same graph as the original data. You may need to vary the colors of the two curves to make sure they both show up on the graph. Comment on what you see. What is happening when you set the Fourier coefficients to zero?
e) Modify your program so that it sets all but the first 2% of the coefficients to zero and run it again.

Exercise 7.5: If you have not done Exercise 7.4 you should do it before this one.

The function $f(t)$ represents a square-wave with amplitude 1 and frequency 1 Hz:

$$f(t) = \begin{cases} 1 & \text{if } \lfloor 2t \rfloor \text{ is even,} \\ -1 & \text{if } \lfloor 2t \rfloor \text{ is odd,} \end{cases} \tag{1}$$

where $\lfloor x \rfloor$ means x rounded down to the next lowest integer. Let us attempt to smooth this function using a Fourier transform, as we did in the previous exercise. Write a program that creates an array of $N = 1000$ elements containing a thousand equally spaced samples from a single cycle of this square-wave. Calculate the discrete Fourier transform of the array. Now set all but the first ten Fourier coefficients to zero, then invert the Fourier transform again to recover the smoothed signal. Make a plot of the result and on the same axes show the original square-wave as well. You should find that the signal is not simply smoothed—there are artifacts, wiggles, in the results. Explain briefly where these come from.

Artifacts similar to these arise when Fourier coefficients are discarded in audio and visual compression schemes like those described in Section 7.3.1 and are the primary source of imperfections in digitally compressed sound and images.

Exercise 7.6: Comparison of the DFT and DCT

This exercise will be easier if you have already done Exercise 7.4.

Exercise 7.4 looked at data representing the variation of the Dow Jones Industrial Average, colloquially called "the Dow," over time. The particular time period studied in that exercise was special in one sense: the value of the Dow at the end of the period was almost the same as at the start, so the function was, roughly speaking, periodic. In the on-line resources there is another file called `dow2.txt`, which also contains data on the Dow but for a different time period, from 2004 until 2008. Over this period the value changed considerably from a starting level around 9000 to a final level around 14000.

a) Write a program similar to the one for Exercise 7.4, part (e), in which you read the data in the file `dow2.txt` and plot it on a graph. Then smooth the data by calculating its Fourier transform, setting all but the first 2% of the coefficients to zero, and inverting the transform again, plotting the result on the same graph as the original data. As in Exercise 7.4 you should see that the data are smoothed, but now there will be an additional artifact. At the beginning and end of the plot you should see large deviations away from the true smoothed function. These occur because the function is required to be periodic—its last value must be the same as its first—so it needs to deviate substantially from the correct value to make the two ends of the function meet. In some situations (including this one) this behavior is unsatisfactory. If we want to use the Fourier transform for smoothing, we would certainly prefer that it not introduce artifacts of this kind.
b) Modify your program to repeat the same analysis using discrete cosine transforms. You can use the functions from `dcst.py` to perform the transforms if you wish. Again discard all but the first 2% of the coefficients, invert the transform, and plot the result. You should see a significant improvement, with less distortion of the function at the ends of the interval. This occurs because, as discussed at the end of Section 7.3, the cosine transform does not force the value of the function to be the same at both ends.

It is because of the artifacts introduced by the strict periodicity of the DFT that the cosine transform is favored for many technological applications, such as audio compression. The artifacts can degrade the sound quality of compressed audio and the cosine transform generally gives better results.

The cosine transform is not wholly free of artifacts itself however. It's true it does not force the function to be periodic, but it does force the gradient to be zero at the ends of the interval (which the ordinary Fourier transform does not). You may be able to see this in your calculations for part (b) above. Look closely at the smoothed function and you should see that its slope is flat at the beginning and end of the interval. The distortion of the function introduced is less than the distortion in part (a), but it's there all the same. To reduce this effect, audio compression schemes often use overlapped cosine transforms, in which transforms are performed on overlapping blocks of samples, so that the portions at the ends of blocks, where the worst artifacts lie, need not be used.

Exercise 7.7: Fast Fourier transform

Write your own program to compute the fast Fourier transform for the case where N is a power of two, based on the formulas given in Section 7.4.1. As a test of your program, use it to calculate the Fourier transform of the data in the file `pitch.txt`, which can be found in the on-line resources. A plot of the data is shown in Fig. 7.3. You should be able to duplicate the results for the Fourier transform shown in Fig. 7.4.

This exercise is quite tricky. You have to calculate the coefficients $E_k^{(m,j)}$ from Eq. (7.43) for all levels m, which means that first you will have to plan how the coefficients will be stored. Since, as we have seen, there are exactly N of them at every level, one way to do it would be to create a two-dimensional complex array of size $N \times (1 + \log_2 N)$, so that it has N complex numbers for each level from zero to $\log_2 N$. Then within level m you have 2^m individual transforms denoted by $j = 0 \ldots 2^m - 1$, each with $N/2^m$ coefficients indexed by k. A simple way to arrange the coefficients would be to put all the $k = 0$ coefficients in a block one after another, then all the $k = 1$ coefficients, and so forth. Then $E_k^{(m,j)}$ would be stored in the $j + 2^m k$ element of the array.

This method has the advantage of being quite simple to program, but the disadvantage of using up a lot of memory space. The array contains $N \log_2 N$ complex numbers, and a complex number typically takes sixteen bytes of memory to store. So if you had to do a large Fourier transform of, say, $N = 10^8$ numbers, it would take $16N \log_2 N \simeq 42$ gigabytes of memory, which is much more than most computers have.

An alternative approach is to notice that we do not really need to store all of the coefficients. At any one point in the calculation we only need the coefficients at the current level and the previous level (from which the current level is calculated). If one is clever one can write a program that uses only two arrays, one for the current level and one for the previous level, each consisting of N complex numbers. Then our transform of 10^8 numbers would require less than four gigabytes, which is fine on most computers.

(There is a third way of storing the coefficients that is even more efficient. If you store the coefficients in the correct order, then you can arrange things so that every time you compute a coefficient for the next level, it gets stored in the same place as the old coefficient from the previous level from which it was calculated, and which you no longer need. With this way of doing things you only need one array of N complex numbers—we say the transform is done "in place." Unfortunately, this in-place Fourier transform is much harder to work out and harder to program. If you are feeling particularly ambitious you might want to give it a try, but it's not for the faint-hearted.)

Exercise 7.8: Diffraction gratings

Exercise 5.19 (page 206) looked at the physics of diffraction gratings, calculating the intensity of the diffraction patterns they produce from the equation

$$I(x) = \left| \int_{-w/2}^{w/2} \sqrt{q(u)}\, \mathrm{e}^{\mathrm{i}2\pi x u/\lambda f}\, \mathrm{d}u \right|^2,$$

where w is the width of the grating, λ is the wavelength of the light, f is the focal length of the lens used to focus the image, and $q(u)$ is the intensity transmission function of the diffraction grating at a distance u from the central axis, i.e., the fraction of the incident light that the grating lets through. In Exercise 5.19 we evaluated this expression directly using standard methods for performing integrals, but a more efficient way to do the calculation is to note that the integral is in fact just a Fourier transform. Approximating the integral, as we did in Eq. (7.13), using the trapezoidal rule, with N points $u_n = nw/N - w/2$, we get

$$\begin{aligned} \int_{-w/2}^{w/2} \sqrt{q(u)}\, \mathrm{e}^{\mathrm{i}2\pi x u/\lambda f}\, \mathrm{d}u &\simeq \frac{w}{N} \mathrm{e}^{\mathrm{i}\pi w x/\lambda f} \sum_{n=0}^{N-1} \sqrt{q(u_n)}\, \mathrm{e}^{\mathrm{i}2\pi w x n/\lambda f N} \\ &= \frac{w}{N} \mathrm{e}^{\mathrm{i}\pi k} \sum_{n=0}^{N-1} y_n\, \mathrm{e}^{\mathrm{i}2\pi k n/N}, \end{aligned}$$

where $k = wx/\lambda f$ and $y_n = \sqrt{q(u_n)}$. Comparing with Eq. (7.15), we see that the sum in this expression is equal to the complex conjugate c_k^* of the kth coefficient of the DFT of y_n. Substituting into the expression for the intensity $I(x)$, we then have

$$I(x_k) = \frac{w^2}{N^2} |c_k|^2,$$

where

$$x_k = \frac{\lambda f}{w} k.$$

Thus we can calculate the intensity of the diffraction pattern at the points x_k by performing a Fourier transform.

There is a catch, however. Given that k is an integer, $k = 0 \ldots N-1$, the points x_k at which the intensity is evaluated have spacing $\lambda f/w$ on the screen. This spacing can be large in some cases, giving us only a rather coarse picture of the diffraction pattern. For instance, in Exercise 5.19 we had $\lambda = 500\,\mathrm{nm}$, $f = 1\,\mathrm{m}$, and $w = 200\,\mu\mathrm{m}$, and the screen was 10 cm wide, which means that $\lambda f/w = 2.5\,\mathrm{mm}$ and we have only forty points on the screen. This is not enough to make a usable plot of the diffraction pattern.

One way to fix this problem is to increase the width of the grating from the given value w to a larger value $W > w$, which makes the spacing $\lambda f/W$ of the points on the screen closer. We can add the extra width on one or the other side of the grating, or both, as we prefer, but—and this is crucial—the extra portion added must be opaque, it must not transmit light, so that the physics of the system does not change. In other words, we need to "pad out" the data points y_n that measure the transmission profile of the grating with additional zeros so as to make the grating wider while keeping its transmission properties the same. For example, to increase the width to $W = 10w$, we would increase the number N of points y_n by a factor of ten, with the extra points set to zero. The extra points can be at the beginning, at the end, or split between the two—it will make no difference to the answer. Then the intensity is given by

$$I(x_k) = \frac{W^2}{N^2}|c_k|^2,$$

where

$$x_k = \frac{\lambda f}{W}k.$$

Write a Python program that uses a fast Fourier transform to calculate the diffraction pattern for a grating with transmission function $q(u) = \sin^2 \alpha u$ (the same as in Exercise 5.19), with slits of width $20\,\mu$m [meaning that $\alpha = \pi/(20\,\mu\text{m})$] and parameters as above: $w = 200\,\mu$m, $W = 10w = 2$ mm, incident light of wavelength $\lambda = 500$ nm, a lens with focal length of 1 meter, and a screen 10 cm wide. Choose a suitable number of points to give a good approximation to the grating transmission function and then make a graph of the diffraction intensity on the screen as a function of position x in the range $-5\,\text{cm} \leq x \leq 5\,\text{cm}$. If you previously did Exercise 5.19, check to make sure your answers to the two exercises agree.

Exercise 7.9: Image deconvolution

You've probably seen it on TV, in one of those crime drama shows. They have a blurry photo of a crime scene and they click a few buttons on the computer and magically the photo becomes sharp and clear, so you can make out someone's face, or some lettering on a sign. Surely (like almost everything else on such TV shows) this is just science fiction? Actually, no. It's not. It's real and in this exercise you'll write a program that does it.

When a photo is blurred each point on the photo gets smeared out according to some "smearing distribution," which is technically called a *point spread function*. We can represent this smearing mathematically as follows. For simplicity let's assume we're working with a black and white photograph, so that the picture can be represented by a single function $a(x, y)$ which tells you the brightness at each point (x, y). And let us denote the point spread function by $f(x, y)$. This means that a single bright dot at the origin ends up appearing as $f(x, y)$ instead. If $f(x, y)$ is a broad function then the picture is badly blurred. If it is a narrow peak then the picture is relatively sharp.

In general the brightness $b(x, y)$ of the blurred photo at point (x, y) is given by

$$b(x, y) = \int_0^K \int_0^L a(x', y') f(x - x', y - y')\, \mathrm{d}x'\, \mathrm{d}y',$$

where $K \times L$ is the dimension of the picture. This equation is called the *convolution* of the picture with the point spread function.

Working with two-dimensional functions can get complicated, so to get the idea of how the math works, let's switch temporarily to a one-dimensional equivalent of our problem. Once we work out the details in 1D we'll return to the 2D version. The one-dimensional version of the convolution above would be

$$b(x) = \int_0^L a(x') f(x - x')\, \mathrm{d}x'.$$

The function $b(x)$ can be represented by a Fourier series as in Eq. (7.5):

$$b(x) = \sum_{k=-\infty}^{\infty} \tilde{b}_k \exp\left(\mathrm{i}\frac{2\pi kx}{L}\right),$$

where

$$\tilde{b}_k = \frac{1}{L}\int_0^L b(x)\exp\left(-\mathrm{i}\frac{2\pi kx}{L}\right)\mathrm{d}x$$

are the Fourier coefficients. Substituting for $b(x)$ in this equation gives

$$\begin{aligned}\tilde{b}_k &= \frac{1}{L}\int_0^L\int_0^L a(x')f(x-x')\exp\left(-\mathrm{i}\frac{2\pi kx}{L}\right)\mathrm{d}x'\,\mathrm{d}x \\ &= \frac{1}{L}\int_0^L\int_0^L a(x')f(x-x')\exp\left(-\mathrm{i}\frac{2\pi k(x-x')}{L}\right)\exp\left(-\mathrm{i}\frac{2\pi kx'}{L}\right)\mathrm{d}x'\,\mathrm{d}x.\end{aligned}$$

Now let us change variables to $X = x - x'$, and we get

$$\tilde{b}_k = \frac{1}{L}\int_0^L a(x')\exp\left(-\mathrm{i}\frac{2\pi kx'}{L}\right)\int_{-x'}^{L-x'} f(X)\exp\left(-\mathrm{i}\frac{2\pi kX}{L}\right)\mathrm{d}X\,\mathrm{d}x'.$$

If we make $f(x)$ a periodic function in the standard fashion by repeating it infinitely many times to the left and right of the interval from 0 to L, then the second integral above can be written as

$$\begin{aligned}\int_{-x'}^{L-x'} f(X)\exp\left(-\mathrm{i}\frac{2\pi kX}{L}\right)\mathrm{d}X &= \int_{-x'}^{0} f(X)\exp\left(-\mathrm{i}\frac{2\pi kX}{L}\right)\mathrm{d}X \\ &\qquad + \int_0^{L-x'} f(X)\exp\left(-\mathrm{i}\frac{2\pi kX}{L}\right)\mathrm{d}X \\ &= \exp\left(\mathrm{i}\frac{2\pi kL}{L}\right)\int_{L-x'}^{L} f(X)\exp\left(-\mathrm{i}\frac{2\pi kX}{L}\right)\mathrm{d}X + \int_0^{L-x'} f(X)\exp\left(-\mathrm{i}\frac{2\pi kX}{L}\right)\mathrm{d}X \\ &= \int_0^L f(X)\exp\left(-\mathrm{i}\frac{2\pi kX}{L}\right)\mathrm{d}X,\end{aligned}$$

which is simply L times the Fourier transform $\tilde{f}_k$ of $f(x)$. Substituting this result back into our equation for $\tilde{b}_k$ we then get

$$\tilde{b}_k = \int_0^L a(x')\exp\left(-\mathrm{i}\frac{2\pi kx'}{L}\right)\tilde{f}_k\,\mathrm{d}x' = L\,\tilde{a}_k\tilde{f}_k.$$

In other words, apart from the factor of L, the Fourier transform of the blurred photo is the product of the Fourier transforms of the unblurred photo and the point spread function.

Now it is clear how we deblur our picture. We take the blurred picture and Fourier transform it to get $\tilde{b}_k = L\,\tilde{a}_k\tilde{f}_k$. We also take the point spread function and Fourier transform it to get $\tilde{f}_k$. Then we divide one by the other:

$$\frac{\tilde{b}_k}{L\tilde{f}_k} = \tilde{a}_k$$

which gives us the Fourier transform of the *unblurred* picture. Then, finally, we do an inverse Fourier transform on $\tilde{a}_k$ to get back the unblurred picture. This process of recovering the unblurred picture from the blurred one, of reversing the convolution process, is called *deconvolution*.

Real pictures are two-dimensional, but the mathematics follows through exactly the same. For a picture of dimensions $K \times L$ we find that the two-dimensional Fourier transforms are related by

$$\tilde{b}_{kl} = KL\tilde{a}_{kl}\tilde{f}_{kl}\,,$$

and again we just divide the blurred Fourier transform by the Fourier transform of the point spread function to get the Fourier transform of the unblurred picture.

In the digital realm of computers, pictures are not pure functions $f(x,y)$ but rather grids of samples, and our Fourier transforms are discrete transforms not continuous ones. But the math works out the same again.

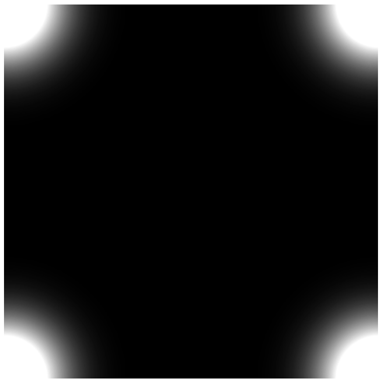
The main complication with deblurring in practice is that we don't usually know the point spread function. Typically we have to experiment with different ones until we find something that works. For many cameras it's a reasonable approximation to assume the point spread function is Gaussian:

$$f(x,y) = \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right),$$

where σ is the width of the Gaussian. Even with this assumption, however, we still don't know the value of σ and we may have to experiment to find a value that works well. In the following exercise, for simplicity, we'll assume we know the value of σ.

a) On the web site you will find a file called `blur.txt` that contains a grid of values representing brightness on a black-and-white photo—a badly out-of-focus one that has been deliberately blurred using a Gaussian point spread function of width $\sigma = 25$. Write a program that reads the grid of values into a two-dimensional array of real numbers and then draws the values on the screen of the computer as a density plot. You should see the photo appear. If you get something wrong it might be upside-down. Work with the details of your program until you get it appearing correctly. (Hint: The picture has the sky, which is bright, at the top and the ground, which is dark, at the bottom.)

b) Write another program that creates an array, of the same size as the photo, containing a grid of samples from drawn from the Gaussian $f(x,y)$ above with $\sigma = 25$. Make a density plot of these values on the screen too, so that you get a visualization of your point spread function. Remember that the point spread function is periodic (along both axes), which means that the values for negative x and y are repeated at the end of the interval. Since the Gaussian is centered on the origin, this means there should be bright patches in each of the four corners of your picture, something like this:



c) Combine your two programs and add Fourier transforms using the functions `rfft2` and `irfft2` from `numpy.fft`, to make a program that does the following:

    i) Reads in the blurred photo
    ii) Calculates the point spread function
    iii) Fourier transforms both
    iv) Divides one by the other
    v) Performs an inverse transform to get the unblurred photo
    vi) Displays the unblurred photo on the screen

    When you are done, you should be able to make out the scene in the photo, although probably it will still not be perfectly sharp.

    Hint: One thing you'll need to deal with is what happens when the Fourier transform of the point spread function is zero, or close to zero. In that case if you divide by it you'll get an error (because you can't divide by zero) or just a very large number (because you're dividing by something small). A workable compromise is that if a value in the Fourier transform of the point spread function is smaller than a certain amount ϵ you don't divide by it—just leave that coefficient alone. The value of ϵ is not very critical but a reasonable value seems to be 10^{-3}.

d) Bearing in mind this last point about zeros in the Fourier transform, what is it that limits our ability to deblur a photo? Why can we not perfectly unblur any photo and make it completely sharp?

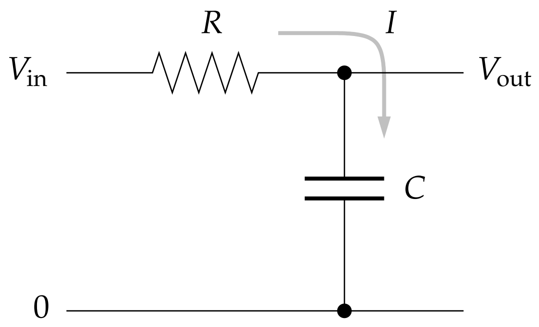
We have seen this process in action here for a normal snapshot, but it is also used in many physics applications where one takes photos. For instance, it is used in astronomy to enhance photos taken by telescopes. It was famously used with images from the Hubble Space Telescope after it was realized that the telescope's main mirror had a serious manufacturing flaw and was returning blurry photos—scientists managed to partially correct the blurring using Fourier transform techniques.

Computational Physics

Exercises for Chapter 8

Exercise 8.1: A low-pass filter

Here is a simple electronic circuit with one resistor and one capacitor:



This circuit acts as a low-pass filter: you send a signal in on the left and it comes out filtered on the right.

Using Ohm's law and the capacitor law and assuming that the output load has very high impedance, so that a negligible amount of current flows through it, we can write down the equations governing this circuit as follows. Let I be the current that flows through R and into the capacitor, and let Q be the charge on the capacitor. Then:

$$IR = V_{\text{in}} - V_{\text{out}}, \qquad Q = CV_{\text{out}}, \qquad I = \frac{\mathrm{d}Q}{\mathrm{d}t}.$$

Substituting the second equation into the third, then substituting the result into the first equation, we find that $V_{\text{in}} - V_{\text{out}} = RC\,(\mathrm{d}V_{\text{out}}/\mathrm{d}t)$, or equivalently

$$\frac{\mathrm{d}V_{\text{out}}}{\mathrm{d}t} = \frac{1}{RC}\big(V_{\text{in}} - V_{\text{out}}\big).$$

a) Write a program (or modify a previous one) to solve this equation for $V_{\text{out}}(t)$ using the fourth-order Runge–Kutta method when in the input signal is a square-wave with frequency 1 and amplitude 1:

$$V_{\text{in}}(t) = \begin{cases} 1 & \text{if } \lfloor 2t \rfloor \text{ is even,} \\ -1 & \text{if } \lfloor 2t \rfloor \text{ is odd,} \end{cases} \tag{1}$$

where $\lfloor x \rfloor$ means x rounded down to the next lowest integer. Use the program to make plots of the output of the filter circuit from $t = 0$ to $t = 10$ when $RC = 0.01$, 0.1, and 1, with initial condition $V_{\text{out}}(0) = 0$. You will have to make a decision about what value of h to use in your calculation. Small values give more accurate results, but the program will take longer to run. Try a variety of different values and choose one for your final calculations that seems sensible to you.

b) Based on the graphs produced by your program, describe what you see and explain what the circuit is doing.

A program similar to the one you wrote is running inside most stereos and music players, to create the effect of the "bass" control. In the old days, the bass control on a stereo would have been connected to a real electronic low-pass filter in the amplifier circuitry, but these days there is just a computer processor that simulates the behavior of the filter in a manner similar to your program.

Exercise 8.2: The Lotka–Volterra equations

The Lotka–Volterra equations are a mathematical model of predator–prey interactions between biological species. Let two variables x and y be proportional to the size of the populations of two species, traditionally called "rabbits" (the prey) and "foxes" (the predators). You could think of x and y as being the population in thousands, say, so that $x = 2$ means there are 2000 rabbits. Strictly the only allowed values of x and y would then be multiples of 0.001, since you can only have whole numbers of rabbits or foxes. But 0.001 is a pretty close spacing of values, so it's a decent approximation to treat x and y as continuous real numbers so long as neither gets very close to zero.

In the Lotka–Volterra model the rabbits reproduce at a rate proportional to their population, but are eaten by the foxes at a rate proportional to both their own population and the population of foxes:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \alpha x - \beta xy,$$

where α and β are constants. At the same time the foxes reproduce at a rate proportional the rate at which they eat rabbits—because they need food to grow and reproduce—but also die of old age at a rate proportional to their own population:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \gamma xy - \delta y,$$

where γ and δ are also constants.

a) Write a program to solve these equations using the fourth-order Runge–Kutta method for the case $\alpha = 1$, $\beta = \gamma = 0.5$, and $\delta = 2$, starting from the initial condition $x = y = 2$. Have the program make a graph showing both x and y as a function of time on the same axes from $t = 0$ to $t = 30$. (Hint: Notice that the differential equations in this case do not depend explicitly on time t—in vector notation, the right-hand side of each equation is a function $f(\mathbf{r})$ with no t dependence. You may nonetheless find it convenient to define a Python function `f(r,t)` including the time variable, so that your program takes the same form as programs given earlier in this chapter. You don't have to do it that way, but it can avoid some confusion. Several of the following exercises have a similar lack of explicit time-dependence.)

b) Describe in words what is going on in the system, in terms of rabbits and foxes.

Exercise 8.3: The Lorenz equations

One of the most celebrated sets of differential equations in physics is the Lorenz equations:

$$\frac{dx}{dt} = \sigma(y - x), \qquad \frac{dy}{dt} = rx - y - xz, \qquad \frac{dz}{dt} = xy - bz,$$

where σ, r, and b are constants. (The names σ, r, and b are odd, but traditional—they are always used in these equations for historical reasons.)

These equations were first studied by Edward Lorenz in 1963, who derived them from a simplified model of weather patterns. The reason for their fame is that they were one of the first incontrovertible examples of *deterministic chaos*, the occurrence of apparently random motion even though there is no randomness built into the equations. We encountered a different example of chaos in the logistic map of Exercise 3.6.

a) Write a program to solve the Lorenz equations for the case $\sigma = 10$, $r = 28$, and $b = \frac{8}{3}$ in the range from $t = 0$ to $t = 50$ with initial conditions $(x, y, z) = (0, 1, 0)$. Have your program make a plot of y as a function of time. Note the unpredictable nature of the motion. (Hint: If you base your program on previous ones, be careful. This problem has parameters r and b with the same names as variables in previous programs—make sure to give your variables new names, or use different names for the parameters, to avoid introducing errors into your code.)

b) Modify your program to produce a plot of z against x. You should see a picture of the famous "strange attractor" of the Lorenz equations, a lop-sided butterfly-shaped plot that never repeats itself.

Exercise 8.4: Building on the results from Example 8.6 above, calculate the motion of a nonlinear pendulum as follows.

a) Write a program to solve the two first-order equations, Eqs. (8.45) and (8.46), using the fourth-order Runge–Kutta method for a pendulum with a 10 cm arm. Use your program to calculate the angle θ of displacement for several periods of the pendulum when it is released from a standstill at $\theta = 179°$ from the vertical. Make a graph of θ as a function of time.

b) Extend your program to create an animation of the motion of the pendulum. Your animation should, at a minimum, include a representation of the moving pendulum bob and the pendulum arm. (Hint: You will probably find the function `rate` discussed in Section 3.5 useful for making your animation run at a sensible speed. Also, you may want to make the step size for your Runge–Kutta calculation smaller than the frame-rate of your animation, i.e., do several Runge–Kutta steps per frame on screen. This is certainly allowed and may help to make your calculation more accurate.)

For a bigger challenge, take a look at Exercise 8.15 on page 398, which invites you to write a program to calculate the chaotic motion of the double pendulum.

Exercise 8.5: The driven pendulum

A pendulum like the one in Exercise 8.4 can be driven by, for example, exerting a small oscillating force horizontally on the mass. Then the equation of motion for the pendulum becomes

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{\ell}\sin\theta + C\cos\theta\sin\Omega t,$$

where C and Ω are constants.

a) Write a program to solve this equation for θ as a function of time with $\ell = 10\,\mathrm{cm}$, $C = 2\,\mathrm{s}^{-2}$ and $\Omega = 5\,\mathrm{s}^{-1}$ and make a plot of θ as a function of time from $t = 0$ to $t = 100\,\mathrm{s}$. Start the pendulum at rest with $\theta = 0$ and $\mathrm{d}\theta/\mathrm{d}t = 0$.

b) Now change the value of Ω, while keeping C the same, to find a value for which the pendulum resonates with the driving force and swings widely from side to side. Make a plot for this case also.

Exercise 8.6: Harmonic and anharmonic oscillators

The simple harmonic oscillator arises in many physical problems, in mechanics, electricity and magnetism, and condensed matter physics, among other areas. Consider the standard oscillator equation

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\omega^2 x.$$

a) Using the methods described in the preceding section, turn this second-order equation into two coupled first-order equations. Then write a program to solve them for the case $\omega = 1$ in the range from $t = 0$ to $t = 50$. A second-order equation requires two initial conditions, one on x and one on its derivative. For this problem use $x = 1$ and $\mathrm{d}x/\mathrm{d}t = 0$ as initial conditions. Have your program make a graph showing the value of x as a function of time.

b) Now increase the amplitude of the oscillations by making the initial value of x bigger—say $x = 2$—and confirm that the period of the oscillations stays roughly the same.

c) Modify your program to solve for the motion of the anharmonic oscillator described by the equation

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -\omega^2 x^3.$$

Again take $\omega = 1$ and initial conditions $x = 1$ and $\mathrm{d}x/\mathrm{d}t = 0$ and make a plot of the motion of the oscillator. Again increase the amplitude. You should observe that the oscillator oscillates faster at higher amplitudes. (You can try lower amplitudes too if you like, which should be slower.) The variation of frequency with amplitude in an anharmonic oscillator was studied previously in Exercise 5.10.

d) Modify your program so that instead of plotting x against t, it plots $\mathrm{d}x/\mathrm{d}t$ against x, i.e., the "velocity" of the oscillator against its "position." Such a plot is called a *phase space* plot.

e) The *van der Pol oscillator*, which appears in electronic circuits and in laser physics, is described by the equation

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} - \mu(1-x^2)\frac{\mathrm{d}x}{\mathrm{d}t} + \omega^2 x = 0.$$

Modify your program to solve this equation from $t = 0$ to $t = 20$ and hence make a phase space plot for the van der Pol oscillator with $\omega = 1$, $\mu = 1$, and initial conditions $x = 1$ and $\mathrm{d}x/\mathrm{d}t = 0$. Try it also for $\mu = 2$ and $\mu = 4$ (still with $\omega = 1$). Make sure you use a small enough value of the time interval h to get a smooth, accurate phase space plot.

Exercise 8.7: Trajectory with air resistance

Many elementary mechanics problems deal with the physics of objects moving or flying through the air, but they almost always ignore friction and air resistance to make the equations solvable. If we're using a computer, however, we don't need solvable equations.

Consider, for instance, a spherical cannonball shot from a cannon standing on level ground. The air resistance on a moving sphere is a force in the opposite direction to the motion with magnitude

$$F = \tfrac{1}{2}\pi R^2 \rho C v^2,$$

where R is the sphere's radius, ρ is the density of air, v is the velocity, and C is the so-called *coefficient of drag* (a property of the shape of the moving object, in this case a sphere).

a) Starting from Newton's second law, $F = ma$, show that the equations of motion for the position (x, y) of the cannonball are

$$\ddot{x} = -\frac{\pi R^2 \rho C}{2m}\,\dot{x}\sqrt{\dot{x}^2 + \dot{y}^2}, \qquad \ddot{y} = -g - \frac{\pi R^2 \rho C}{2m}\,\dot{y}\sqrt{\dot{x}^2 + \dot{y}^2},$$

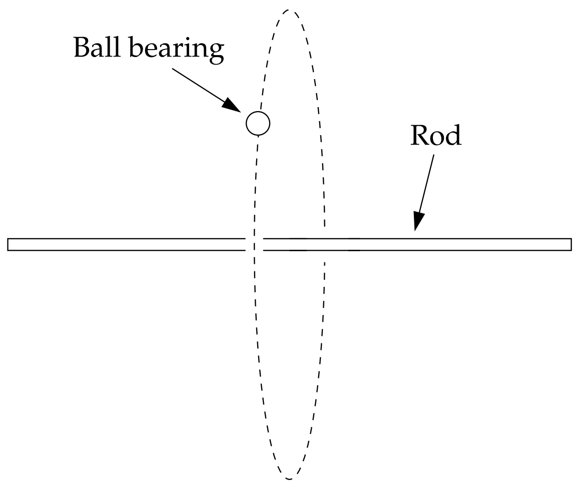
where m is the mass of the cannonball, g is the acceleration due to gravity, and $\dot{x}$ and $\ddot{x}$ are the first and second derivatives of x with respect to time.

b) Change these two second-order equations into four first-order equations using the methods you have learned, then write a program that solves the equations for a cannonball of mass 1 kg and radius 8 cm, shot at 30° to the horizontal with initial velocity $100\,\mathrm{m\,s^{-1}}$. The density of air is $\rho = 1.22\,\mathrm{kg\,m^{-3}}$ and the coefficient of drag for a sphere is $C = 0.47$. Make a plot of the trajectory of the cannonball (i.e., a graph of y as a function of x).

c) When one ignores air resistance, the distance traveled by a projectile does not depend on the mass of the projectile. In real life, however, mass certainly does make a difference. Use your program to estimate the total distance traveled (over horizontal ground) by the cannonball above, and then experiment with the program to determine whether the cannonball travels further if it is heavier or lighter. You could, for instance, plot a series of trajectories for cannonballs of different masses, or you could make a graph of distance traveled as a function of mass. Describe briefly what you discover.

Exercise 8.8: Space garbage

A heavy steel rod and a spherical ball-bearing, discarded by a passing spaceship, are floating in zero gravity and the ball bearing is orbiting around the rod under the effect of its gravitational pull:



For simplicity we'll assume that the rod is of negligible cross-section and heavy enough that it doesn't move significantly, and that the ball bearing is orbiting around the rod's mid-point in a plane perpendicular to the rod.

a) Treating the rod as a line of mass M and length L and the ball bearing as a point mass m, show that the attractive force F felt by the ball bearing in the direction toward the center of the rod is given by

$$F = \frac{GMm}{L}\sqrt{x^2+y^2}\int_{-L/2}^{L/2}\frac{\mathrm{d}z}{(x^2+y^2+z^2)^{3/2}},$$

where G is Newton's gravitational constant and x and y are the coordinates of the ball bearing in the plane perpendicular to the rod. The integral can be done in closed form and gives

$$F = \frac{GMm}{\sqrt{(x^2+y^2)(x^2+y^2+L^2/4)}}.$$

Hence show that the equations of motion for the position x, y of the ball bearing in the xy-plane are

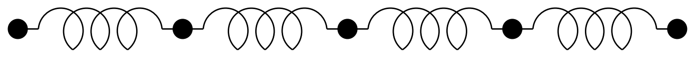
$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -GM\frac{x}{r^2\sqrt{r^2+L^2/4}}, \qquad \frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -GM\frac{y}{r^2\sqrt{r^2+L^2/4}},$$

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

b) Convert these two second-order equations into four first-order ones using the techniques of Section 8.3. Then, working in units where $G = 1$, write a program to solve them for $M = 10$, $L = 2$, and initial conditions $(x, y) = (1, 0)$ with velocity of $+1$ in the y direction. Calculate the orbit from $t = 0$ to $t = 10$ and make a plot of it, meaning a plot of y against x. You should find that the ball bearing does not orbit in a circle or ellipse as a planet does, but has a precessing orbit, which arises because the attractive force is not a simple $1/r^2$ force as it is for a planet orbiting the Sun.

Exercise 8.9: Vibration in a one-dimensional system

In Example 6.2 on page 235 we studied the motion of a system of N identical masses (in zero gravity) joined by identical linear springs like this:



As we showed, the horizontal displacements ξ_i of masses $i = 1 \dots N$ satisfy equations of motion

$$\begin{aligned} m\frac{d^2\xi_1}{dt^2} &= k(\xi_2 - \xi_1) + F_1, \\ m\frac{d^2\xi_i}{dt^2} &= k(\xi_{i+1} - \xi_i) + k(\xi_{i-1} - \xi_i) + F_i, \\ m\frac{d^2\xi_N}{dt^2} &= k(\xi_{N-1} - \xi_N) + F_N. \end{aligned}$$

where m is the mass, k is the spring constant, and F_i is the external force on mass i. In Example 6.2 we showed how these equations could be solved by guessing a form for the solution and using a matrix method. Here we'll solve them more directly.

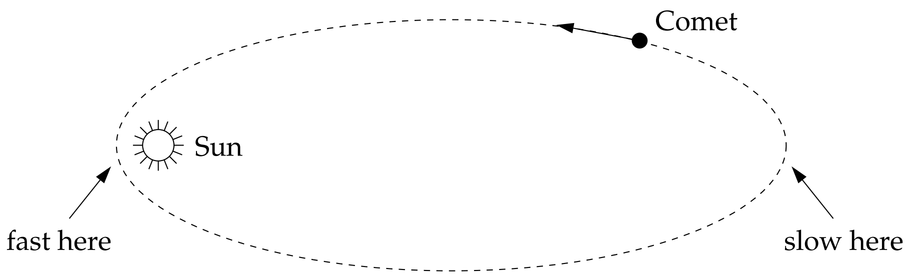
a) Write a program to solve for the motion of the masses using the fourth-order Runge–Kutta method for the case we studied previously where $m = 1$ and $k = 6$, and the driving forces are all zero except for $F_1 = \cos \omega t$ with $\omega = 2$. Plot your solutions for the displacements ξ_i of all the masses as a function of time from $t = 0$ to $t = 20$ on the same plot. Write your program to work with general N, but test it out for small values—$N = 5$ is a reasonable choice.

   You will need first of all to convert the N second-order equations of motion into $2N$ first-order equations. Then combine all of the dependent variables in those equations into a single large vector $\mathbf{r}$ to which you can apply the Runge–Kutta method in the standard fashion.

b) Modify your program to create an animation of the movement of the masses, represented as spheres on the computer screen. You will probably find the `rate` function discussed in Section 3.5 useful for making your animation run at a sensible speed.

Exercise 8.10: Cometary orbits

Many comets travel in highly elongated orbits around the Sun. For much of their lives they are far out in the solar system, moving very slowly, but on rare occasions their orbit brings them close to the Sun for a fly-by and for a brief period of time they move very fast indeed:



This is a classic example of a system for which an adaptive step size method is useful, because for the large periods of time when the comet is moving slowly we can use long time-steps, so that the program runs quickly, but short time-steps are crucial in the brief but fast-moving period close to the Sun.

The differential equation obeyed by a comet is straightforward to derive. The force between the Sun, with mass M at the origin, and a comet of mass m with position vector $\mathbf{r}$ is GMm/r^2 in direction $-\mathbf{r}/r$ (i.e., the direction towards the Sun), and hence Newton's second law tells us that

$$m\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -\left(\frac{GMm}{r^2}\right)\frac{\mathbf{r}}{r}.$$

Canceling the m and taking the x component we have

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -GM\frac{x}{r^3},$$

and similarly for the other two coordinates. We can, however, throw out one of the coordinates because the comet stays in a single plane as it orbits. If we orient our axes so that this plane is perpendicular to the z-axis, we can forget about the z coordinate and we are left with just two second-order equations to solve:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -GM\frac{x}{r^3}, \qquad \frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -GM\frac{y}{r^3},$$

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

a) Turn these two second-order equations into four first-order equations, using the methods you have learned.

b) Write a program to solve your equations using the fourth-order Runge–Kutta method with a *fixed* step size. You will need to look up the mass of the Sun and Newton's gravitational constant G. As an initial condition, take a comet at coordinates $x = 4$ billion kilometers and $y = 0$ (which is somewhere out around the orbit of Neptune) with initial velocity $v_x = 0$ and $v_y = 500\,\mathrm{m\,s^{-1}}$. Make a graph showing the trajectory of the comet (i.e., a plot of y against x).

   Choose a fixed step size h that allows you to accurately calculate at least two full orbits of the comet. Since orbits are periodic, a good indicator of an accurate calculation is that successive orbits of the comet lie on top of one another on your plot. If they do not then you need a smaller value of h. Give a short description of your findings. What value of h did you use? What did you observe in your simulation? How long did the calculation take?

c) Make a copy of your program and modify the copy to do the calculation using an adaptive step size. Set a target accuracy of $\delta = 1$ kilometer per year in the position of the comet and again plot the trajectory. What do you see? How do the speed, accuracy, and step size of the calculation compare with those in part (b)?

d) Modify your program to place dots on your graph showing the position of the comet at each Runge–Kutta step around a single orbit. You should see the steps getting closer together when the comet is close to the Sun and further apart when it is far out in the solar system.

Calculations like this can be extended to cases where we have more than one orbiting body—see Exercise 8.16 for an example. We can include planets, moons, asteroids, and others. Analytic calculations are impossible for such complex systems, but with careful numerical solution of differential equations we can calculate the motions of objects throughout the entire solar system.

Exercise 8.11: Write a program to solve the differential equation

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} - \left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2 + x + 5 = 0$$

using the leapfrog method. Solve from $t = 0$ to $t = 50$ in steps of $h = 0.001$ with initial condition $x = 1$ and $\mathrm{d}x/\mathrm{d}t = 0$. Make a plot of your solution showing x as a function of t.

Exercise 8.12: Orbit of the Earth

Use the Verlet method to calculate the orbit of the Earth around the Sun. The equations of motion for the position $\mathbf{r} = (x, y)$ of the planet in its orbital plane are the same as those for any orbiting body and are derived in Exercise 8.10 on page 361. In vector form, they are

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -GM\frac{\mathbf{r}}{r^3},$$

where $G = 6.6738 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$ is Newton's gravitational constant and $M = 1.9891 \times 10^{30}\,\mathrm{kg}$ is the mass of the Sun.

The orbit of the Earth is not perfectly circular, the planet being sometimes closer to and sometimes further from the Sun. When it is at its closest point, or *perihelion*, it is moving precisely tangentially (i.e., perpendicular to the line between itself and the Sun) and it has distance $1.4710 \times 10^{11}\,\mathrm{m}$ from the Sun and linear velocity $3.0287 \times 10^4\,\mathrm{m\,s^{-1}}$.

a) Write a program to calculate the orbit of the Earth using the Verlet method, Eqs. (8.77) and (8.78), with a time-step of $h = 1$ hour. Make a plot of the orbit, showing several complete revolutions about the Sun. The orbit should be very slightly, but visibly, non-circular.

b) The gravitational potential energy of the Earth is $-GMm/r$, where $m = 5.9722 \times 10^{24}\,\mathrm{kg}$ is the mass of the planet, and its kinetic energy is $\frac{1}{2}mv^2$ as usual. Modify your program to calculate both of these quantities at each step, along with their sum (which is the total energy), and make a plot showing all three as a function of time on the same axes. You should find that the potential and kinetic energies vary visibly during the course of an orbit, but the total energy remains constant.

c) Now plot the total energy alone without the others and you should be able to see a slight variation over the course of an orbit. Because you're using the Verlet method, however, which conserves energy in the long term, the energy should always return to its starting value at the end of each complete orbit.

Exercise 8.13: Planetary orbits

This exercise asks you to calculate the orbits of two of the planets using the Bulirsch–Stoer method. The method gives results significantly more accurate than the Verlet method used to calculate the Earth's orbit in Exercise 8.12.

The equations of motion for the position x, y of a planet in its orbital plane are the same as those for any orbiting body and are derived in Exercise 8.10 on page 361:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -GM\frac{x}{r^3}, \qquad \frac{\mathrm{d}^2y}{\mathrm{d}t^2} = -GM\frac{y}{r^3},$$

where $G = 6.6738 \times 10^{-11}\,\mathrm{m^3\,kg^{-1}\,s^{-2}}$ is Newton's gravitational constant, $M = 1.9891 \times 10^{30}\,\mathrm{kg}$ is the mass of the Sun, and $r = \sqrt{x^2 + y^2}$.

Let us first solve these equations for the orbit of the Earth, duplicating the results of Exercise 8.12, though with greater accuracy. The Earth's orbit is not perfectly circular, but rather slightly elliptical. When it is at its closest approach to the Sun, its perihelion, it is moving precisely tangentially (i.e., perpendicular to the line between itself and the Sun) and it has distance $1.4710 \times 10^{11}\,\mathrm{m}$ from the Sun and linear velocity $3.0287 \times 10^4\,\mathrm{ms^{-1}}$.

a) Write a program, or modify the one from Example 8.7, to calculate the orbit of the Earth using the Bulirsch–Stoer method to a positional accuracy of 1 km per year. Divide the orbit into intervals of length $H = 1$ week and then calculate the solution for each interval using the combined modified midpoint/Richardson extrapolation method described in this section. Make a plot of the orbit, showing at least one complete revolution about the Sun.

b) Modify your program to calculate the orbit of the dwarf planet Pluto. The distance between the Sun and Pluto at perihelion is $4.4368 \times 10^{12}\,\mathrm{m}$ and the linear velocity is $6.1218 \times 10^3\,\mathrm{ms^{-1}}$. Choose a suitable value for H to make your calculation run in reasonable time, while once again giving a solution accurate to 1 km per year.

   You should find that the orbit of Pluto is significantly elliptical—much more so than the orbit of the Earth. Pluto is a Kuiper belt object, similar to a comet, and (unlike true planets) it's typical for such objects to have quite elliptical orbits.

Exercise 8.14: Quantum oscillators

Consider the one-dimensional, time-independent Schrödinger equation in a harmonic (i.e., quadratic) potential $V(x) = V_0x^2/a^2$, where V_0 and a are constants.

a) Write down the Schrödinger equation for this problem and convert it from a second-order equation to two first-order ones, as in Example 8.9. Write a program, or modify the one from Example 8.9, to find the energies of the ground state and the first two excited states for these equations when m is the electron mass, $V_0 = 50\,\mathrm{eV}$, and $a = 10^{-11}\,\mathrm{m}$. Note that in theory the wavefunction goes all the way out to $x = \pm\infty$, but you can get good answers by using a large but finite interval. Try using $x = -10a$ to $+10a$, with the wavefunction $\psi = 0$ at both boundaries. (In effect, you are putting the harmonic oscillator in a box with impenetrable walls.) The wavefunction is real everywhere, so you don't need to use complex variables, and you can use evenly spaced points for the solution—there is no need to use an adaptive method for this problem.

The quantum harmonic oscillator is known to have energy states that are equally spaced. Check that this is true, to the precision of your calculation, for your answers. (Hint: The ground state has energy in the range 100 to 200 eV.)

b) Now modify your program to calculate the same three energies for the anharmonic oscillator with $V(x) = V_0 x^4/a^4$, with the same parameter values.

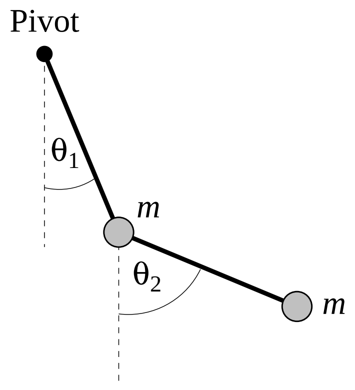
c) Modify your program further to calculate the properly normalized wavefunctions of the anharmonic oscillator for the three states and make a plot of them, all on the same axes, as a function of x over a modest range near the origin—say $x = -5a$ to $x = 5a$.

To normalize the wavefunctions you will have to evaluate the integral $\int_{-\infty}^{\infty} |\psi(x)|^2 \,dx$ and then rescale ψ appropriately to ensure that the area under the square of each of the wavefunctions is 1. Either the trapezoidal rule or Simpson's rule will give you a reasonable value for the integral. Note, however, that you may find a few very large values at the end of the array holding the wavefunction. Where do these large values come from? Are they real, or spurious?

One simple way to deal with the large values is to make use of the fact that the system is symmetric about its midpoint and calculate the integral of the wavefunction over only the left-hand half of the system, then double the result. This neatly misses out the large values.

Exercise 8.15: The double pendulum

If you did Exercise 8.4 you will have created a program to calculate the movement of a nonlinear pendulum. Although it is nonlinear, the nonlinear pendulum's movement is nonetheless perfectly regular and periodic—there are no surprises. A *double pendulum*, on the other hand, is completely the opposite—chaotic and unpredictable. A double pendulum consists of a normal pendulum with another pendulum hanging from its end. For simplicity let us ignore friction, and assume that both pendulums have bobs of the same mass m and massless arms of the same length ℓ. Thus the setup looks like this:



The position of the arms at any moment in time is uniquely specified by the two angles θ_1 and θ_2. The equations of motion for the angles are most easily derived using the Lagrangian formalism, as follows.

The heights of the two bobs, measured from the level of the pivot are

$$h_1 = -\ell \cos\theta_1, \qquad h_2 = -\ell(\cos\theta_1 + \cos\theta_2),$$

so the potential energy of the system is

$$V = mgh_1 + mgh_2 = -mg\ell(2\cos\theta_1 + \cos\theta_2),$$

where g is the acceleration due to gravity. The (linear) velocities of the two bobs are given by

$$v_1 = \ell\dot\theta_1, \qquad v_2^2 = \ell^2\bigl[\dot\theta_1^2 + \dot\theta_2^2 + 2\dot\theta_1\dot\theta_2\cos(\theta_1-\theta_2)\bigr],$$

where $\dot\theta$ means the derivative of θ with respect to time t. (If you don't see where the second velocity equation comes from, it's a good exercise to derive it for yourself from the geometry of the pendulum.) Now the total kinetic energy is

$$T = \tfrac12 mv_1^2 + \tfrac12 mv_2^2 = m\ell^2\bigl[\dot\theta_1^2 + \tfrac12\dot\theta_2^2 + \dot\theta_1\dot\theta_2\cos(\theta_1-\theta_2)\bigr],$$

and the Lagrangian of the system is

$$\mathscr{L} = T - V = m\ell^2\bigl[\dot\theta_1^2 + \tfrac12\dot\theta_2^2 + \dot\theta_1\dot\theta_2\cos(\theta_1-\theta_2)\bigr] + mg\ell(2\cos\theta_1 + \cos\theta_2).$$

Then the equations of motion are given by the Euler–Lagrange equations

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial\mathscr{L}}{\partial\dot\theta_1}\right) = \frac{\partial\mathscr{L}}{\partial\theta_1}, \qquad \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial\mathscr{L}}{\partial\dot\theta_2}\right) = \frac{\partial\mathscr{L}}{\partial\theta_2},$$

which in this case give

$$\begin{aligned} 2\ddot\theta_1 + \ddot\theta_2\cos(\theta_1-\theta_2) + \dot\theta_2^2\sin(\theta_1-\theta_2) + 2\frac{g}{\ell}\sin\theta_1 &= 0, \\ \ddot\theta_2 + \ddot\theta_1\cos(\theta_1-\theta_2) - \dot\theta_1^2\sin(\theta_1-\theta_2) + \frac{g}{\ell}\sin\theta_2 &= 0, \end{aligned}$$

where the mass m has canceled out.

These are second-order equations, but we can convert them into first-order ones by the usual method, defining two new variables, ω_1 and ω_2, thus:

$$\dot\theta_1 = \omega_1, \qquad \dot\theta_2 = \omega_2.$$

In terms of these variables our equations of motion become

$$\begin{aligned} 2\dot\omega_1 + \dot\omega_2\cos(\theta_1-\theta_2) + \omega_2^2\sin(\theta_1-\theta_2) + 2\frac{g}{\ell}\sin\theta_1 &= 0, \\ \dot\omega_2 + \dot\omega_1\cos(\theta_1-\theta_2) - \omega_1^2\sin(\theta_1-\theta_2) + \frac{g}{\ell}\sin\theta_2 &= 0. \end{aligned}$$

Finally we have to rearrange these into the standard form of Eq. (8.29) with a single derivative on the left-hand side of each one, which gives

$$\dot{\omega}_1 = -\frac{\omega_1^2 \sin(2\theta_1 - 2\theta_2) + 2\omega_2^2 \sin(\theta_1 - \theta_2) + (g/\ell)\bigl[\sin(\theta_1 - 2\theta_2) + 3\sin\theta_1\bigr]}{3 - \cos(2\theta_1 - 2\theta_2)},$$

$$\dot{\omega}_2 = \frac{4\omega_1^2 \sin(\theta_1 - \theta_2) + \omega_2^2 \sin(2\theta_1 - 2\theta_2) + 2(g/\ell)\bigl[\sin(2\theta_1 - \theta_2) - \sin\theta_2\bigr]}{3 - \cos(2\theta_1 - 2\theta_2)}.$$

(This last step is quite tricky and involves some trigonometric identities. If you're not certain of how the calculation goes you may find it useful to go through the derivation for yourself.)

These two equations, along with the equations $\dot{\theta}_1 = \omega_1$ and $\dot{\theta}_2 = \omega_2$, give us four first-order equations which between them define the motion of the double pendulum.

a) Derive an expression for the total energy $E = T + V$ of the system in terms of the variables θ_1, θ_2, ω_1, and ω_2, plus the constants g, ℓ, and m.

b) Write a program using the fourth-order Runge–Kutta method to solve the equations of motion for the case where $\ell = 40\,\text{cm}$, with the initial conditions $\theta_1 = \theta_2 = 90°$ and $\omega_1 = \omega_2 = 0$. Use your program to calculate the total energy of the system assuming that the mass of the bobs is 1 kg each, and make a graph of energy as a function of time from $t = 0$ to $t = 100$ seconds.

   Because of energy conservation, the total energy should be constant over time (actually it should be zero for this particular set of initial conditions), but you will find that it is not perfectly constant because of the approximate nature of the solution of the differential equation. Choose a suitable value of the step size h to ensure that the variation in energy is less than 10^{-5} Joules over the course of the calculation.

c) Make a copy of your program and modify the copy to create a second program that does not produce a graph, but instead makes an animation of the motion of the double pendulum over time. At a minimum, the animation should show the two arms and the two bobs.

   Hint: As in Exercise 8.4 you will probably find the function `rate` useful in order to make your program run at a steady speed. You will probably also find that the value of h needed to get the required accuracy in your solution gives a frame-rate much faster than any that can reasonably be displayed in your animation, so you won't be able to display every time-step of the calculation in the animation. Instead you will have to arrange the program so that it updates the animation only once every several Runge–Kutta steps.

Exercise 8.16: The three-body problem

If you mastered Exercise 8.10 on cometary orbits, here's a more challenging problem in celestial mechanics—and a classic in the field—the *three-body problem*.

Three stars, in otherwise empty space, are initially at rest, with the following masses and positions, in arbitrary units:

	Mass	x	y
Star 1	150	3	1
Star 2	200	-1	-2
Star 3	250	-1	1

(All the z coordinates are zero, so the three stars lie in the xy plane.)

a) Show that the equation of motion governing the position $\mathbf{r}_1$ of the first star is

$$\frac{d^2\mathbf{r}_1}{dt^2} = Gm_2\frac{\mathbf{r}_2-\mathbf{r}_1}{|\mathbf{r}_2-\mathbf{r}_1|^3} + Gm_3\frac{\mathbf{r}_3-\mathbf{r}_1}{|\mathbf{r}_3-\mathbf{r}_1|^3}$$

and derive two similar equations for the positions $\mathbf{r}_2$ and $\mathbf{r}_3$ of the other two stars. Then convert the three second-order equations into six equivalent first-order equations, using the techniques you have learned.

b) Working in units where $G = 1$, write a program to solve your equations and hence calculate the motion of the stars from $t = 0$ to $t = 2$. Make a plot showing the trails of all three stars (i.e., a graph of y against x for each star).

c) Modify your program to make an animation of the motion on the screen from $t = 0$ to $t = 10$. You may wish to make the three stars different sizes or colors (or both) so that you can tell which is which.

To do this calculation properly you will need to use an adaptive step size method, for the same reasons as in Exercise 8.10—the stars move very rapidly when they are close together and very slowly when they are far apart. An adaptive method is the only way to get the accuracy you need in the fast-moving parts of the motion without wasting hours uselessly calculating the slow parts with a tiny step size. Construct your program so that it introduces an error of no more than 10^{-3} in the position of any star per unit time.

Creating an animation with an adaptive step size can be challenging, since the steps do not all correspond to the same amount of real time. The simplest thing to do is just to ignore the varying step sizes and make an animation as if they were all equal, updating the positions of the stars on the screen at every step or every several steps. This will give you a reasonable visualization of the motion, but it will look a little odd because the stars will slow down, rather than speed up, as they come close together, because the adaptive calculation will automatically take more steps in this region.

A better solution is to vary the frame-rate of your animation so that the frames run proportionally faster when h is smaller, meaning that the frame-rate needs to be equal to C/h for some constant C. You can achieve this by using the `rate` function from the `visual` package to set a different frame-rate on each step, equal to C/h. If you do this, it's a good idea to not let the value of h grow too large, or the animation will make some large jumps that look uneven on the screen. Insert extra program lines to ensure that h never exceeds a value $h_{\max}$ that you choose. Values for the constants of around $C = 0.1$ and $h_{\max} = 10^{-3}$ seem to give reasonable results.

Exercise 8.17: Cometary orbits and the Bulirsch–Stoer method

Repeat the calculation of the cometary orbit in Exercise 8.10 (page 361) using the adaptive Bulirsch–Stoer method of Section 8.5.6 to calculate a solution accurate to $\delta = 1$ kilometer per year in the position of the comet. Calculate the solution from $t = 0$ to $t = 2 \times 10^9$ s, initially using just a single time interval of size $H = 2 \times 10^9$ s and allowing a maximum of $n = 8$ modified midpoint steps before dividing the interval in half and trying again. Then these intervals may be subdivided again, as described in Section 8.5.6, as many times as necessary until the method converges in eight steps or less in each interval.

Make a plot of the orbit (i.e., a plot of y against x) and have your program add dots to the trajectory to show where the ends of the time intervals lie. You should see the time intervals getting shorter in the part of the trajectory close to the Sun, where the comet is moving rapidly.

Hint: The simplest way to do this calculation is to make use of recursion, the ability of a Python function to call itself. (If you're not familiar with the idea of recursion you might like to look at Exercise 2.13 on page 83 before doing this exercise.) Write a user-defined function called, say, `step(r,t,H)` that takes as arguments the position vector $\mathbf{r} = (x, y)$ at a starting time t and an interval length H, and returns the new value of $\mathbf{r}$ at time $t + H$. This function should perform the modified midpoint/Richardson extrapolation calculation described in Section 8.5.5 until either the calculation converges to the required accuracy or you reach the maximum number $n = 8$ of modified midpoint steps. If it fails to converge in eight steps, have your function call itself, twice, to calculate separately the solution for the first then the second half of the interval from t to $t + H$, something like this:

```
r1 = step(r,t,H/2)
r2 = step(r1,t+H/2,H/2)
```

(Then *these* functions can call themselves, and so forth, subdividing the interval as many times as necessary to reach the required accuracy.)

Exercise 8.18: Oscillating chemical reactions

The *Belousov–Zhabotinsky reaction* is a chemical oscillator, a cocktail of chemicals which, when heated, undergoes a series of reactions that cause the chemical concentrations in the mixture to oscillate between two extremes. You can add an indicator dye to the reaction which changes color depending on the concentrations and watch the mixture switch back and forth between two different colors for as long as you go on heating the mixture.

Physicist Ilya Prigogine formulated a mathematical model of this type of chemical oscillator, which he called the "Brusselator" after his home town of Brussels. The equations for the Brusselator are

$$\frac{\mathrm{d}x}{\mathrm{d}t} = 1 - (b+1)x + ax^2y, \qquad \frac{\mathrm{d}y}{\mathrm{d}t} = bx - ax^2y.$$

Here x and y represent concentrations of chemicals and a and b are positive constants.

Write a program to solve these equations for the case $a = 1$, $b = 3$ with initial conditions $x = y = 0$, to an accuracy of at least $\delta = 10^{-10}$ per unit time in both x and y, using the adaptive Bulirsch–Stoer method described in Section 8.5.6. Calculate a solution from $t = 0$ to $t = 20$, initially using a single time interval of size $H = 20$. Allow a maximum of $n = 8$ modified midpoint steps in an interval before you divide in half and try again.

Make a plot of your solutions for x and y as a function of time, both on the same graph, and have your program add dots to the curves to show where the boundaries of the time intervals lie. You should find that the points are significantly closer together in parts of the solution where the variables are changing rapidly.

Hint: The simplest way to perform the calculation is to make use of recursion, as described in Exercise 8.17.

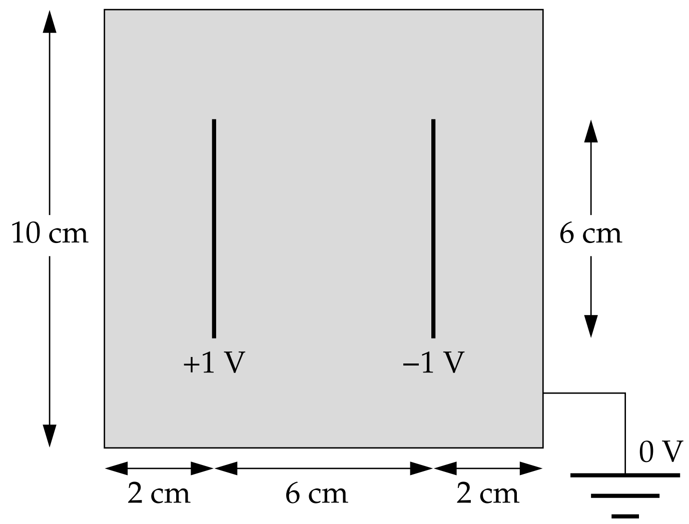
Computational Physics

Exercises for Chapter 9

Exercise 9.1: Write a program, or modify the one from Example 9.1, to solve Poisson's equation for the system described in Example 9.2. Work in units where $\epsilon_0 = 1$ and continue the iteration until your solution for the electric potential changes by less than 10^{-6} V per step at every grid point.

Exercise 9.2: Use the Gauss–Seidel method to solve Laplace's equation for the two-dimensional problem in Example 9.1—a square box 1 m on each side, at voltage $V = 1$ volt along the top wall and zero volts along the other three. Use a grid of spacing $a = 1$ cm, so that there are 100 grid points along each wall, or 101 if you count the points at both ends. Continue the iteration of the method until the value of the electric potential changes by no more than $\delta = 10^{-6}$ V at any grid point on any step, then make a density plot of the final solution, similar to that shown in Fig. 9.3. Experiment with different values of ω to find which value gives the fastest solution. As mentioned above, you should find that a value around 0.9 does well. In general larger values cause the calculation to run faster, but if you choose too large a value the speed drops off and for values above 1 the calculation becomes unstable.

Exercise 9.3: Consider the following simple model of an electronic capacitor, consisting of two flat metal plates enclosed in a square metal box:



For simplicity let us model the system in two dimensions. Using any of the methods we have studied, write a program to calculate the electrostatic potential in the box on a grid of 100×100 points, where the walls of the box are at voltage zero and the two plates (which are of negligible thickness) are at voltages ± 1 V as shown. Have your program calculate the value of the potential at each grid point to a precision of 10^{-6} volts and then make a density plot of the result.

Hint: Notice that the capacitor plates are at fixed *voltage*, not fixed charge, so this problem differs from the problem with the two charges in Exercise 9.1. In effect, the capacitor plates are part of the boundary condition in this case: they behave the same way as the walls of the box, with potentials that are fixed at a certain value and cannot change.

Exercise 9.4: Thermal diffusion in the Earth's crust

A classic example of a diffusion problem with a time-varying boundary condition is the diffusion of heat into the crust of the Earth, as surface temperature varies with the seasons. Suppose the mean daily temperature at a particular point on the surface varies as:

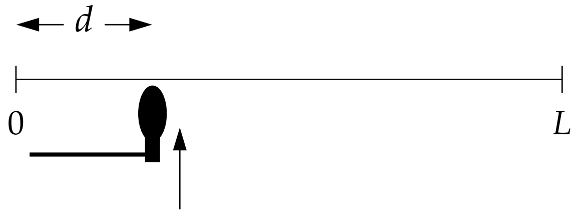
$$T_0(t) = A + B\sin\frac{2\pi t}{\tau},$$

where $\tau = 365$ days, $A = 10°\text{C}$ and $B = 12°\text{C}$. At a depth of 20 m below the surface almost all annual temperature variation is ironed out and the temperature is, to a good approximation, a constant 11°C (which is higher than the mean surface temperature of 10°C—temperature increases with depth, due to heating from the hot core of the planet). The thermal diffusivity of the Earth's crust varies somewhat from place to place, but for our purposes we will treat it as constant with value $D = 0.1\,\text{m}^2\,\text{day}^{-1}$.

Write a program, or modify one of the ones given in this chapter, to calculate the temperature profile of the crust as a function of depth up to 20 m and time up to 10 years. Start with temperature everywhere equal to 10°C, except at the surface and the deepest point, choose values for the number of grid points and the time-step h, then run your program for the first nine simulated years, to allow it to settle down into whatever pattern it reaches. Then for the tenth and final year plot four temperature profiles taken at 3-month intervals on a single graph to illustrate how the temperature changes as a function of depth and time.

Exercise 9.5: FTCS solution of the wave equation

Consider a piano string of length L, initially at rest. At time $t = 0$ the string is struck by the piano hammer a distance d from the end of the string:



The string vibrates as a result of being struck, except at the ends, $x = 0$ and $x = L$, where it is held fixed.

a) Write a program that uses the FTCS method to solve the complete set of simultaneous first-order equations, Eq. (9.28), for the case $v = 100\,\text{ms}^{-1}$, with the initial condition that $\phi(x) = 0$ everywhere but the velocity $\psi(x)$ is nonzero, with profile

$$\psi(x) = C\frac{x(L-x)}{L^2}\exp\left[-\frac{(x-d)^2}{2\sigma^2}\right],$$

where $L = 1\,\text{m}$, $d = 10\,\text{cm}$, $C = 1\,\text{ms}^{-1}$, and $\sigma = 0.3\,\text{m}$. You will also need to choose a value for the time-step h. A reasonable choice is $h = 10^{-6}\,\text{s}$.

b) Make an animation of the motion of the piano string using the facilities provided by the `visual` package, which we studied in Section 3.4. There are various ways you could do this. A simple one would be to just place a small sphere at the location of each grid point on the string. A more sophisticated approach would be to use the `curve` object in the `visual` package—see the on-line documentation at `www.vpython.org` for details. A convenient feature of the curve object is that you can specify its set of x positions and y positions separately as arrays. In this exercise the x positions only need to specified once, since they never change, while the y positions will need to be specified anew each time you take a time-step. Also, since the vertical displacement of the string is much less than its horizontal length, you will probably need to multiply the vertical displacement by a fairly large factor to make it visible on the screen.

   Allow your animation to run for some time, until numerical instabilities start to appear.

Exercise 9.6: What would the equivalent of Eq. (9.7) be in three dimensions?

Exercise 9.7: The relaxation method for ordinary differential equations

There is no reason why the relaxation method must be restricted to the solution of differential equations with two or more independent variables. It can also be applied to those with one independent variable, i.e., to ordinary differential equations. In this context, as with partial differential equations, it is a technique for solving boundary value problems, which are less common with ordinary differential equations but do occur—we discussed them in Section 8.6.

Consider the problem we looked at in Example 8.8 on page 390, in which a ball of mass $m = 1\,\text{kg}$ is thrown from height $x = 0$ into the air and lands back at $x = 0$ ten seconds later. The problem is to calculate the trajectory of the ball, but we cannot do it using initial value methods like the ordinary Runge–Kutta method because we are not told the initial velocity of the ball. One approach to finding a solution is the shooting method of Section 8.6.1. Another is the relaxation method.

Ignoring friction effects, the trajectory is the solution of the ordinary differential equation

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -g,$$

where g is the acceleration due to gravity.

a) Replacing the second derivative in this equation with its finite-difference approximation, Eq. (5.109), derive a relaxation-method equation for solving this problem on a time-like "grid" of points with separation h.

b) Taking the boundary conditions to be that $x = 0$ at $t = 0$ and $t = 10$, write a program to solve for the height of the ball as a function of time using the relaxation method with 100 points and make a plot of the result from $t = 0$ to $t = 10$. Run the relaxation method until the answers change by 10^{-6} or less at every point on each step.

Note that, unlike the shooting method, the relaxation method does not give us the initial value of the velocity needed to achieve the required solution. It gives us only the solution itself, although one could get an approximation to the initial velocity by calculating a numerical derivative of the solution at time $t = 0$. On balance, however, the relaxation method for ordinary differential equations is most useful when one wants to know the details of the solution itself, but not the initial conditions needed to achieve it.

Exercise 9.8: The Schrödinger equation and the Crank–Nicolson method

Perhaps the most important partial differential equation, at least for physicists, is the Schrödinger equation. This exercise uses the Crank–Nicolson method to solve the full time-dependent Schrödinger equation and hence develop a picture of how a wavefunction evolves over time. The following exercise, Exercise 9.9, solves the same problem again, but using the spectral method.

We will look at the Schrödinger equation in one dimension. The techniques for calculating solutions in two or three dimensions are basically the same as for one dimension, but the calculations take much longer on the computer, so in the interests of speed we'll stick with one dimension. In one dimension the Schrödinger equation for a particle of mass M with no potential energy reads

$$-\frac{\hbar^2}{2M}\frac{\partial^2\psi}{\partial x^2} = \mathrm{i}\hbar\frac{\partial\psi}{\partial t}.$$

For simplicity, let's put our particle in a box with impenetrable walls, so that we only have to solve the equation in a finite-sized space. The box forces the wavefunction ψ to be zero at the walls, which we'll put at $x = 0$ and $x = L$.

Replacing the second derivative in the Schrödinger equation with a finite difference and applying Euler's method, we get the FTCS equation

$$\psi(x,t+h) = \psi(x,t) + h\frac{\mathrm{i}\hbar}{2ma^2}\big[\psi(x+a,t) + \psi(x-a,t) - 2\psi(x,t)\big],$$

where a is the spacing of the spatial grid points and h is the size of the time-step. (Be careful not to confuse the time-step h with Planck's constant $\hbar$.) Performing a similar step in reverse, we get the implicit equation

$$\psi(x,t+h) - h\frac{\mathrm{i}\hbar}{2ma^2}\big[\psi(x+a,t+h) + \psi(x-a,t+h) - 2\psi(x,t+h)\big] = \psi(x,t).$$

And taking the average of these two, we get the Crank–Nicolson equation for the Schrödinger equation:

$$\begin{aligned}\psi(x,t+h) - h\frac{\mathrm{i}\hbar}{4ma^2}\big[\psi(x+a,t+h) + \psi(x-a,t+h) - 2\psi(x,t+h)\big]&\\ = \psi(x,t) + h\frac{\mathrm{i}\hbar}{4ma^2}\big[\psi(x+a,t) + \psi(x-a,t) - 2\psi(x,t)\big]&.\end{aligned}$$

This gives us a set of simultaneous equations, one for each grid point.

The boundary conditions on our problem tell us that $\psi = 0$ at $x = 0$ and $x = L$ for all t. In between these points we have grid points at a, $2a$, $3a$, and so forth. Let us arrange the values of ψ at these interior points into a vector

$$\boldsymbol{\psi}(t) = \begin{pmatrix} \psi(a,t) \\ \psi(2a,t) \\ \psi(3a,t) \\ \vdots \end{pmatrix}.$$

Then the Crank–Nicolson equations can be written in the form

$$\mathbf{A}\boldsymbol{\psi}(t+h) = \mathbf{B}\boldsymbol{\psi}(t),$$

where the matrices $\mathbf{A}$ and $\mathbf{B}$ are both symmetric and tridiagonal:

$$\mathbf{A} = \begin{pmatrix} a_1 & a_2 & & & \\ a_2 & a_1 & a_2 & & \\ & a_2 & a_1 & a_2 & \\ & & a_2 & a_1 & \\ & & & & \ddots \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} b_1 & b_2 & & & \\ b_2 & b_1 & b_2 & & \\ & b_2 & b_1 & b_2 & \\ & & b_2 & b_1 & \\ & & & & \ddots \end{pmatrix},$$

with

$$a_1 = 1 + h\frac{\mathrm{i}\hbar}{2ma^2}, \qquad a_2 = -h\frac{\mathrm{i}\hbar}{4ma^2}, \qquad b_1 = 1 - h\frac{\mathrm{i}\hbar}{2ma^2}, \qquad b_2 = h\frac{\mathrm{i}\hbar}{4ma^2}.$$

(Note the different signs and the factors of 2 and 4 in the denominators.)

The equation $\mathbf{A}\boldsymbol{\psi}(t+h) = \mathbf{B}\boldsymbol{\psi}(t)$ has precisely the form $\mathbf{A}\mathbf{x} = \mathbf{v}$ of the simultaneous equation problems we studied in Chapter 6 and can be solved using the same methods. Specifically, since the matrix $\mathbf{A}$ is tridiagonal in this case, we can use the fast tridiagonal version of Gaussian elimination that we looked at in Section 6.1.6.

Consider an electron (mass $M = 9.109 \times 10^{-31}$ kg) in a box of length $L = 10^{-8}$ m. Suppose that at time $t = 0$ the wavefunction of the electron has the form

$$\psi(x,0) = \exp\left[-\frac{(x-x_0)^2}{2\sigma^2}\right] \mathrm{e}^{\mathrm{i}\kappa x},$$

where

$$x_0 = \frac{L}{2}, \qquad \sigma = 1 \times 10^{-10}\,\mathrm{m}, \qquad \kappa = 5 \times 10^{10}\,\mathrm{m}^{-1},$$

and $\psi = 0$ on the walls at $x = 0$ and $x = L$. (This expression for $\psi(x,0)$ is not normalized—there should really be an overall multiplying coefficient to make sure that the probability density for the electron integrates to unity. It's safe to drop the constant, however, because the Schrödinger equation is linear, so the constant cancels out on both sides of the equation and plays no part in the solution.)

a) Write a program to perform a single step of the Crank–Nicolson method for this electron, calculating the vector $\boldsymbol{\psi}(t)$ of values of the wavefunction, given the initial wavefunction above and using $N = 1000$ spatial slices with $a = L/N$. Your program will have to perform the following steps. First, given the vector $\boldsymbol{\psi}(0)$ at $t = 0$, you will have to multiply by the matrix $\mathbf{B}$ to get a vector $\mathbf{v} = \mathbf{B}\boldsymbol{\psi}$. Because of the tridiagonal form of $\mathbf{B}$, this is fairly simple. The ith component of $\mathbf{v}$ is given by

$$v_i = b_1\psi_i + b_2(\psi_{i+1} + \psi_{i-1}).$$

You will also have to choose a value for the time-step h. A reasonable choice is $h = 10^{-18}$ s.

Second you will have to solve the linear system $\mathbf{Ax} = \mathbf{v}$ for $\mathbf{x}$, which gives you the new value of $\boldsymbol{\psi}$. You could do this using a standard linear equation solver like the function `solve` in `numpy.linalg`, but since the matrix $\mathbf{A}$ is tridiagonal a better approach would be to use the fast solver for banded matrices given in Appendix E, which can be imported from the file `banded.py` (which you can find in the on-line resources). Note that although the wavefunction of a particle in principle has a complex value, in this case the wavefunction is always real—all the coefficients in the equations above are real numbers so if, as here, the wavefunction starts off real, then it remains real. Thus you do not need to use a complex array to represent the vector $\boldsymbol{\psi}$. A real one will do the job.

Third, once you have the code in place to perform a single step of the calculation, extend your program to perform repeated steps and hence solve for ψ at a sequence of times a separation h apart. Note that the matrix $\mathbf{A}$ is independent of time, so it doesn't change from one step to another. You can set up the matrix just once and then keep on reusing it for every step.

b) Extend your program to make an animation of the solution by displaying the real part of the wavefunction at each time-step. You can use the function `rate` from the package `visual` to ensure a smooth frame-rate for your animation—see Section 3.5 on page 117.

   There are various ways you could do the animation. A simple one would be to just place a small sphere at each grid point with vertical position representing the value of the real part of the wavefunction. A more sophisticated approach would be to use the `curve` object from the `visual` package—see the on-line documentation at `www.vpython.org` for details. Depending on what coordinates you use for measuring x, you may need to scale the values of the wavefunction by an additional constant to make them a reasonable size on the screen. (If you measure your x position in meters then a scale factor of about 10^{-9} works well for the wavefunction.)

c) Run your animation for a while and describe what you see. Write a few sentences explaining in physics terms what is going on in the system.

Exercise 9.9: The Schrödinger equation and the spectral method

This exercise uses the spectral method to solve the time-dependent Schödinger equation

$$-\frac{\hbar^2}{2M}\frac{\partial^2\psi}{\partial x^2} = \mathrm{i}\hbar\frac{\partial\psi}{\partial t}$$

for the same system as in Exercise 9.8, a single particle in one dimension in a box of length L with impenetrable walls. The wavefunction in such a box necessarily goes to zero on the walls and hence one possible (unnormalized) solution of the equation is

$$\psi_k(x,t) = \sin\left(\frac{\pi k x}{L}\right) \mathrm{e}^{\mathrm{i}Et/\hbar},$$

where the energy E can be found by substituting into the Schrödinger equation, giving

$$E = \frac{\pi^2\hbar^2 k^2}{2ML^2}.$$

As with the vibrating string of Section 9.3.4, we can write a full solution as a linear combination of such individual solutions, which on the grid points $x_n = nL/N$ takes the value

$$\psi(x_n,t) = \frac{1}{N}\sum_{k=1}^{N-1} b_k \sin\left(\frac{\pi k n}{N}\right) \exp\left(\mathrm{i}\frac{\pi^2\hbar k^2}{2ML^2}t\right),$$

where the b_k are some set of (possibly complex) coefficients that specify the exact shape of the wavefunction and the leading factor of $1/N$ is optional but convenient.

Since the Schrödinger equation (unlike the wave equation) is first order in time, we need only a single initial condition on the value of $\psi(x,t)$ to specify the coefficients b_k, although, since the coefficients are in general complex, we will need to calculate both real and imaginary parts of each coefficient.

As in Exercise 9.8 we consider an electron (mass $M = 9.109 \times 10^{-31}$ kg) in a box of length $L = 10^{-8}$ m. At time $t = 0$ the wavefunction of the electron has the form

$$\psi(x,0) = \exp\left[-\frac{(x-x_0)^2}{2\sigma^2}\right] \mathrm{e}^{\mathrm{i}\kappa x},$$

where

$$x_0 = \frac{L}{2}, \qquad \sigma = 1\times 10^{-10}\,\mathrm{m}, \qquad \kappa = 5\times 10^{10}\,\mathrm{m}^{-1},$$

and $\psi = 0$ on the walls at $x = 0$ and $x = L$.

a) Write a program to calculate the values of the coefficients b_k, which for convenience can be broken down into their real and imaginary parts as $b_k = \alpha_k + \mathrm{i}\eta_k$. Divide the box into $N = 1000$ slices and create two arrays containing the real and imaginary parts of $\psi(x_n, 0)$ at each grid point. Perform discrete sine transforms on each array separately and hence calculate the values of the α_k and η_k for all $k = 1 \ldots N-1$.

   To perform the discrete sine transforms, you can use the fast transform function `dst` from the package `dcst`, which you can find in the on-line resources in the file named `dcst.py`. A copy of the code for the package can also be found in Appendix E. The function takes an array of N real numbers and returns the discrete sine transform as another array of N numbers.

   (Note that the first element of the input array should in principle always be zero for a sine transform, but if it is not the `dst` function will simply pretend that it is. Similarly the first element of the returned array is always zero, since the $k = 0$ coefficient of a sine transform is always zero. So in effect, the sine transform really only takes $N - 1$ real numbers and transforms them into another $N - 1$ real numbers. In some implementations of the discrete sine transform, therefore, though not the one in the package `dsct` used here, the first element of each array is simply omitted, since it's always zero anyway, and the arrays are only $N - 1$ elements long.)

b) Putting $b_k = \alpha_k + i\eta_k$ in the solution above and taking the real part we get

$$\operatorname{Re} \psi(x_n, t) = \frac{1}{N} \sum_{k=1}^{N-1} \left[\alpha_k \cos\left(\frac{\pi^2 \hbar k^2}{2ML^2} t\right) - \eta_k \sin\left(\frac{\pi^2 \hbar k^2}{2ML^2} t\right) \right] \sin\left(\frac{\pi k n}{N}\right)$$

for the real part of the wavefunction. This is an inverse sine transform with coefficients equal to the quantities in the square brackets. Extend your program to calculate the real part of the wavefunction $\psi(x, t)$ at an arbitrary time t using this formula and the inverse discrete sine transform function `idst`, also from the package `dcst`. Test your program by making a graph of the wavefunction at time $t = 10^{-16}$ s.

c) Extend your program further to make an animation of the wavefunction over time, similar to that described in part (b) of Exercise 9.8 above. A suitable time interval for each frame of the animation is about 10^{-18} s.

d) Run your animation for a while and describe what you see. Write a few sentences explaining in physics terms what is going on in the system.

COMPUTATIONAL PHYSICS

EXERCISES FOR CHAPTER 10

Exercise 10.1: Rolling dice

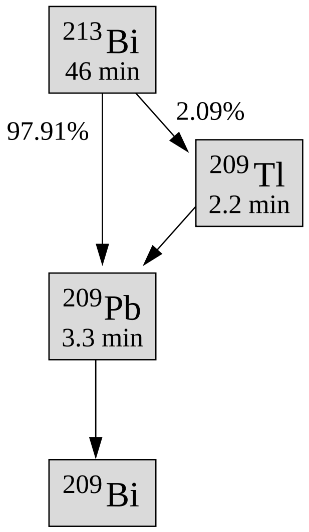
a) Write a program that generates and prints out two random numbers between 1 and 6, to simulate the rolling of two dice.

b) Modify your program to simulate the rolling of two dice a million times and count the number of times you get a double six. Divide by a million to get the *fraction* of times you get a double six. You should get something close to, though probably not exactly equal to, $\frac{1}{36}$.

Exercise 10.2: Radioactive decay chain

This exercise looks at a more advanced version of the simple radioactive decay simulation in Example 10.1.

The isotope ^{213}Bi decays to stable ^{209}Bi via one of two different routes, with probabilities and half-lives thus:



(Technically, ^{209}Bi isn't really stable, but it has a half-life of more than 10^{19} years, a billion times the age of the universe, so it might as well be.)

Starting with a sample consisting of 10 000 atoms of ^{213}Bi, simulate the decay of the atoms as in Example 10.1 by dividing time into slices of length $\delta t = 1$ s each and on each step doing the following:

a) For each atom of ^{209}Pb in turn, decide at random, with the appropriate probability, whether it decays or not. (The probability can be calculated from Eq. (10.3).) Count the total number that decay, subtract it from the number of ^{209}Pb atoms, and add it to the number of ^{209}Bi atoms.

b) Now do the same for ^{209}Tl, except that decaying atoms are subtracted from the total for ^{209}Tl and added to the total for ^{209}Pb.

c) For ^{213}Bi the situation is more complicated: when a ^{213}Bi atom decays you have to decide at random with the appropriate probability the route by which it decays. Count the numbers that decay by each route and add and subtract accordingly.

Note that you have to work up the chain from the bottom like this, not down from the top, to avoid inadvertently making the same atom decay twice on a single step.

Keep track of the number of atoms of each of the four isotopes at all times for 20 000 seconds and make a single graph showing the four numbers as a function of time on the same axes.

Exercise 10.3: Brownian motion

Brownian motion is the motion of a particle, such as a smoke or dust particle, in a gas, as it is buffeted by random collisions with gas molecules. Make a simple computer simulation of such a particle in two dimensions as follows. The particle is confined to a square grid or lattice $L \times L$ squares on a side, so that its position can be represented by two integers $i, j = 0 \ldots L-1$. It starts in the middle of the grid. On each step of the simulation, choose a random direction—up, down, left, or right—and move the particle one step in that direction. This process is called a random walk. The particle is not allowed to move outside the limits of the lattice—if it tries to do so, choose a new random direction to move in.

Write a program to perform a million steps of this process on a lattice with $L = 101$ and make an animation on the screen of the position of the particle. (We choose an odd length for the side of the square so that there is one lattice site exactly in the center.)

Note: The `visual` package doesn't always work well with the `random` package, but if you import functions from `visual` first, then from `random`, you should avoid problems.

Exercise 10.4: Radioactive decay again

Redo the calculation from Example 10.1, but this time using the faster method described in the preceding section. Using the transformation method, generate 1000 random numbers from the nonuniform distribution of Eq. (10.5) to represent the times of decay of 1000 atoms of ^{208}Tl (which has half-life 3.053 minutes). Then make a plot showing the number of atoms that have not decayed as a function of time, i.e., a plot as a function of t showing the number of atoms whose chosen decay times are greater than t.

Hint: You may find it useful to know that the package `numpy` contains a function `sort` that will rearrange the elements of an array in increasing order. That is, "`b = sort(a)`" returns a new array `b` containing the same numbers as `a`, but rearranged in order from smallest to largest.

Exercise 10.5:

a) Write a program to evaluate the integral in Eq. (10.22) using the "hit-or-miss" Monte Carlo method of Section 10.2 with 10 000 points. Also evaluate the error on your estimate.

b) Now estimate the integral again using the mean value method with 10 000 points. Also evaluate the error.

You should find that the error is somewhat smaller using the mean value method.

Exercise 10.6: Construct a general proof that the mean value method always does better, or at least no worse, than the "hit-or-miss" method of Section 10.2, as follows.

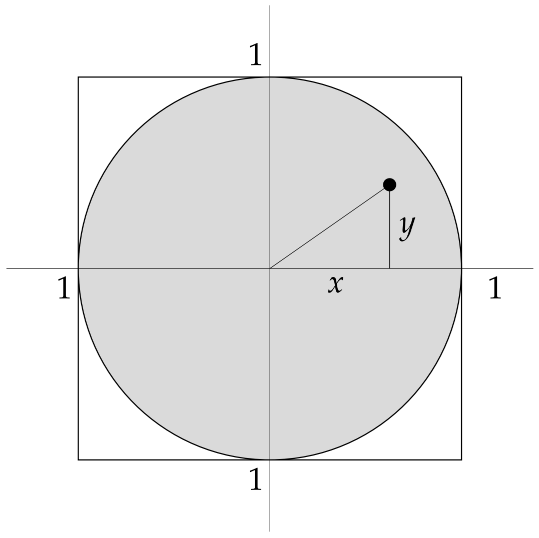
a) For an integral of the form (10.27) with $f(x) \geq 0$ everywhere in the domain of integration, show that Eq. (10.23) can be rewritten as
$$I \simeq (b-a)H\langle s\rangle,$$
where H is the vertical height of the box enclosing the function to be integrated (so that the box's area is $A = (b-a)H$) and $\langle s\rangle$ is the average of variables s_i defined such that $s_i = 1$ if the ith point in the Monte Carlo procedure was a "hit" (it fell below the curve of $f(x)$) and $s_i = 0$ if it was a "miss." Hence argue that the hit-or-miss method will never be more accurate than the mean value method if the variance of f in Eq. (10.32) satisfies $\operatorname{var}(f) \leq H^2 \operatorname{var}(s)$.

b) Show that the variance of a single variable s_i is $\operatorname{var}(s) = p(1-p)$, where $p = I/A$ as in Section 10.2. Show further that $p = \langle f\rangle/H$ and $H^2 \operatorname{var}(s) = \langle f\rangle(H - \langle f\rangle)$ and thus that the hit-or-miss method will never be the more accurate method if $\langle f(f-H)\rangle \leq 0$. Given that the value of $f(x)$ never falls outside the interval from 0 to H, prove that this last condition is always true.

The hit-or-miss method can be extended to the case where $f(x)$ is not always positive by adding a constant onto $f(x)$ large enough to make it always positive, calculating the integral of the resulting function, then subtracting the constant again at the end. The proof above can be extended to this case by noting that the variance of f is unaffected by additive constants, and hence the mean value method is always the more accurate of the two integration methods for any function, positive or not.

Exercise 10.7: Volume of a hypersphere

This exercise asks you to estimate the volume of a sphere of unit radius in ten dimensions using a Monte Carlo method. Consider the equivalent problem in two dimensions, the area of a circle of unit radius:



The area of the circle, the shaded area above, is given by the integral

$$I = \iint_{-1}^{+1} f(x,y)\,\mathrm{d}x\,\mathrm{d}y,$$

where $f(x,y) = 1$ everywhere inside the circle and zero everywhere outside. In other words,

$$f(x,y) = \begin{cases} 1 & \text{if } x^2 + y^2 \le 1, \\ 0 & \text{otherwise.} \end{cases}$$

So if we didn't already know the area of the circle, we could calculate it by Monte Carlo integration. We would generate a set of N random points (x,y), where both x and y are in the range from -1 to 1. Then the two-dimensional version of Eq. (10.33) for this calculation would be

$$I \simeq \frac{4}{N} \sum_{i=1}^{N} f(x_i, y_i).$$

Generalize this method to the ten-dimensional case and write a program to perform a Monte Carlo calculation of the volume of a sphere of unit radius in ten dimensions.

If we had to do a ten-dimensional integral the traditional way, it would take a very long time. Even with only 100 points along each axis (which wouldn't give a very accurate result) we'd still have $100^{10} = 10^{20}$ points to sample, which is impossible on any computer. But using the Monte Carlo method we can get a pretty good result with a million points or so.

Exercise 10.8: Calculate a value for the integral

$$I = \int_0^1 \frac{x^{-1/2}}{\mathrm{e}^x + 1}\,\mathrm{d}x,$$

using the importance sampling formula, Eq. (10.42), with $w(x) = x^{-1/2}$, as follows.

a) Show that the probability distribution $p(x)$ from which the sample points should be drawn is given by

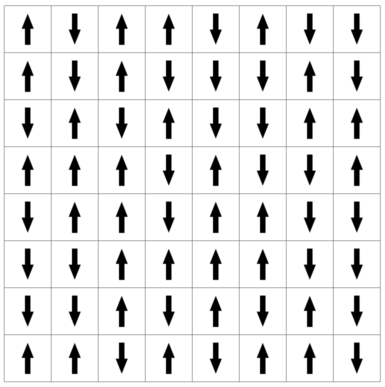
$$p(x) = \frac{1}{2\sqrt{x}}$$

and derive a transformation formula for generating random numbers between zero and one from this distribution.

b) Using your formula, sample $N = 1\,000\,000$ random points and hence evaluate the integral. You should get a value around 0.84.

Exercise 10.9: The Ising model

The Ising model is a theoretical model of a magnet. The magnetization of a magnetic material is made up of the combination of many small magnetic dipoles spread throughout the material. If these dipoles point in random directions then the overall magnetization of the system will be close to zero, but if they line up so that all or most of them point in the same direction then the system can acquire a macroscopic magnetic moment—it becomes magnetized. The Ising model is a model of this process in which the individual moments are represented by dipoles or "spins" arranged on a grid or lattice:



In this case we are using a square lattice in two dimensions, although the model can be defined in principle for any lattice in any number of dimensions.

The spins themselves, in this simple model, are restricted to point in only two directions, up and down. Mathematically the spins are represented by variables $s_i = \pm 1$ on the points of the lattice, $+1$ for up-pointing spins and -1 for down-pointing ones. Dipoles in real magnets can typically point in any spatial direction, not just up or down, but the Ising model, with its restriction to just the two directions, captures a lot of the important physics while being significantly simpler to understand.

Another important feature of many magnetic materials is that the individual dipoles in the material may interact magnetically in such a way that it is energetically favorable for them to line up in the same direction. The magnetic potential energy due to the interaction of two dipoles is proportional to their dot product, but in the Ising model this simplifies to just the product $s_i s_j$ for spins on sites i and j of the lattice, since the spins are one-dimensional scalars, not vectors. Then the actual energy of interaction is $-J s_i s_j$, where J is a positive interaction constant. The minus sign ensures that the interactions are *ferromagnetic*, meaning the energy is lower when dipoles are lined up. A ferromagnetic interaction implies that the material will magnetize if given the chance. (In some materials the interaction has the opposite sign so that the dipoles prefer to be antialigned. Such a material is said to be *antiferromagnetic*, but we will not look at the antiferromagnetic case here.)

Normally it is assumed that spins interact only with those that are immediately adjacent to them on the lattice, which gives a total energy for the entire system equal to

$$E = -J \sum_{\langle ij \rangle} s_i s_j \,,$$

where the notation $\langle ij \rangle$ indicates a sum over pairs i, j that are adjacent on the lattice. On the square lattice we use in this exercise each spin has four adjacent neighbors with which it interacts.

Write a program to perform a Markov chain Monte Carlo simulation of the Ising model on the square lattice for a system of 20×20 spins. You will need to set up variables to hold the value ± 1 of the spin on each lattice site, probably using a two-dimensional integer array, and then take the following steps.

a) First write a function to calculate the total energy of the system, as given by the equation above. That is, for a given array of values of the spins, go through every pair of adjacent spins and add up the contributions $s_i s_j$ from all of them, then multiply by $-J$. Hint 1:

Each unique pair of adjacent spins crops up only once in the sum. Thus there is a term $-Js_1s_2$ if spins 1 and 2 are adjacent to one another, but you do not also need a term $-Js_2s_1$. Hint 2: To make your final program to run in a reasonable amount of time, you will find it helpful if you can work out a way to calculate the energy using Python's ability to do arithmetic with entire arrays at once. If you do the calculation step by step, your program will be significantly slower.

b) Now use your function as the basis for a Metropolis-style simulation of the Ising model with $J = 1$ and temperature $T = 1$ in units where the Boltzmann constant k_B is also 1. Initially set the spin variables randomly to ± 1, so that on average about a half of them are up and a half down, giving a total magnetization of roughly zero. Then choose a spin at random, flip it, and calculate the new energy after it is flipped, and hence also the change in energy as a result of the flip. Then decide whether to accept the flip using the Metropolis acceptance formula, Eq. (10.60). If the move is rejected you will have to flip the spin back to where it was. Otherwise you keep the flipped spin. Now repeat this process for many moves.

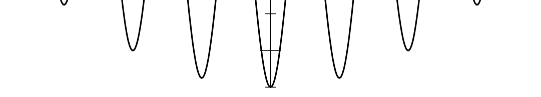
c) Make a plot of the total magnetization $M = \sum_i s_i$ of the system as a function of time for a million Monte Carlo steps. You should see that the system develops a "spontaneous magnetization," a nonzero value of the overall magnetization. Hint: While you are working on your program, do shorter runs, of maybe ten thousand steps at a time. Once you have it working properly, do a longer run of a million steps to get the final results.

d) Run your program several times and observe the sign of the magnetization that develops, positive or negative. Describe what you find and give a brief explanation of what is happening.

e) Make a second version of your program that produces an animation of the system using the `visual` package, with spheres or squares of two colors, on a regular grid, to represent the up and down spins. Run it with temperature $T = 1$ and observe the behavior of the system. Then run it two further times at temperatures $T = 2$ and $T = 3$. Explain briefly what you see in your three runs. How and why does the behavior of the system change as temperature is increased?

Exercise 10.10: Global minimum of a function

Consider the function $f(x) = x^2 - \cos 4\pi x$, which looks like this:



Clearly the global minimum of this function is at $x = 0$.

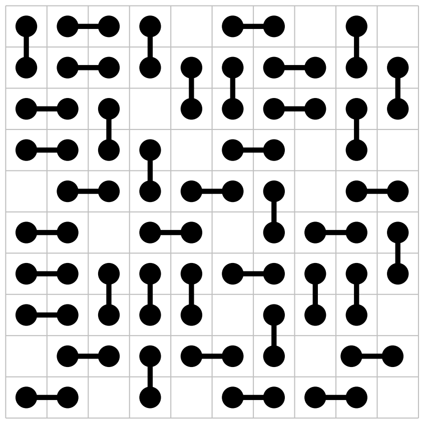
a) Write a program to confirm this fact using simulated annealing starting at, say, $x = 2$, with Monte Carlo moves of the form $x \to x + \delta$ where δ is a random number drawn from a Gaussian distribution with mean zero and standard deviation one. (See Section 10.1.6 for a reminder of how to generate Gaussian random numbers.) Use an exponential cooling schedule and adjust the start and end temperatures, as well as the exponential constant, until you find values that give good answers in reasonable time. Have your program make a plot of the values of x as a function of time during the run and have it print out the final value of x at the end. You will find the plot easier to interpret if you make it using dots rather than lines, with a statement of the form `plot(x,".")` or similar.

b) Now adapt your program to find the minimum of the more complicated function $f(x) = \cos x + \cos \sqrt{2}x + \cos \sqrt{3}x$ in the range $0 < x < 50$.

Hint: The correct answer for part (b) is around $x = 16$, but there are also competing minima around $x = 2$ and $x = 42$ that your program might find. In real-world situations, it is often good enough to find any reasonable solution to a problem, not necessarily the absolute best, so the fact that the program sometimes settles on these other solutions is not necessarily a bad thing.

Exercise 10.11: The dimer covering problem

A well studied problem in condensed matter physics is the *dimer covering problem* in which dimers, meaning polymers with only two atoms, land on the surface of a solid, falling in the spaces between the atoms on the surface and forming a grid like this:



No two dimers are allowed to overlap. The question is how many dimers we can fit in the entire $L \times L$ square. The answer, in this simple case, is clearly $\frac{1}{2}L \times L$, but suppose we did not know this. (There are more complicated versions of the problem on different lattices, or with differently shaped elements, for which the best solution is far from obvious, or in some cases not known at all.)

a) Write a program to solve the problem using simulated annealing on a 50×50 lattice. The "energy" function for the system is *minus* the number of dimers, so that it is minimized when the dimers are a maximum. The moves for the Markov chain are as follows:

i) Choose two adjacent sites on the lattice at random.
ii) If those two sites are currently occupied by a single dimer, remove the dimer from the lattice.
iii) If they are currently both empty, add a dimer.
iv) Otherwise, do nothing.

Perform an animation of the state of the system over time as the simulation runs.

b) Try exponential cooling schedules with different time constants. A reasonable first value to try is $\tau = 10\,000$ steps. For faster cooling schedules you should see that the solutions found are poorer—a smaller fraction of the lattice is filled with dimers and there are larger holes in between them—but for slower schedules the calculation can find quite good, but usually not perfect, coverings of the lattice.

Exercise 10.12: A random point on the surface of the Earth

Suppose you wish to choose a random point on the surface of the Earth. That is, you want to choose a value of the latitude and longitude such that every point on the planet is equally likely to be chosen. In a physics context, this is equivalent to choosing a random vector direction in three-dimensional space (something that one has to do quite often in physics calculations).

Recall that in spherical coordinates θ, ϕ the element of solid angle is $\sin\theta \,\mathrm{d}\theta \,\mathrm{d}\phi$, and the total solid angle in a whole sphere is 4π. Hence the probability of our point falling in a particular element is

$$p(\theta,\phi)\,\mathrm{d}\theta\,\mathrm{d}\phi = \frac{\sin\theta\,\mathrm{d}\theta\,\mathrm{d}\phi}{4\pi}.$$

We can break this up into its θ part and its ϕ part thus:

$$p(\theta,\phi)\,\mathrm{d}\theta\,\mathrm{d}\phi = \frac{\sin\theta\,\mathrm{d}\theta}{2} \times \frac{\mathrm{d}\phi}{2\pi} = p(\theta)\,\mathrm{d}\theta \times p(\phi)\,\mathrm{d}\phi.$$

a) What are the ranges of the variables θ and ϕ? Verify that the two distributions $p(\theta)$ and $p(\phi)$ are correctly normalized—they integrate to 1 over the appropriate ranges.

b) Find formulas for generating angles θ and ϕ drawn from the distributions $p(\theta)$ and $p(\phi)$. (The ϕ one is trivial, but the θ one is not.)

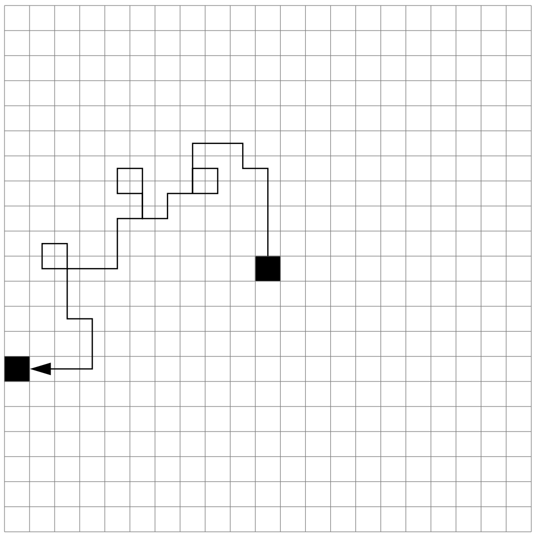
c) Write a program that generates a random θ and ϕ using the formulas you worked out. (Hint: In Python the function `acos` in the `math` package returns the arc cosine in radians of a given number.)

d) Modify your program to generate 500 such random points, convert the angles to x, y, z coordinates assuming the radius of the globe is 1, and then visualize the points in three-dimensional space using the `visual` package with small spheres (of radius, say, 0.02). You should end up with a three-dimensional globe spelled out on the screen in random points.

Exercise 10.13: Diffusion-limited aggregation

This exercise builds upon Exercise 10.3 on page 457. If you have not done that exercise you should do it before doing this one.

In this exercise you will develop a computer program to reproduce one of the most famous models in computational physics, *diffusion-limited aggregation*, or DLA for short. There are various versions of DLA, but the one we'll study is as follows. You take a square grid with a single particle in the middle. The particle performs a random walk from square to square on the grid until it reaches a point on the edge of the system, at which point it "sticks" to the edge, becoming anchored there and immovable:



Then a second particle starts at the center and does a random walk until it sticks either to an edge or to the other particle. Then a third particle starts, and so on. Each particle starts at the center and walks until it sticks either to an edge or to any anchored particle.

a) Make a copy of the Brownian motion program that you wrote for Exercise 10.3. This will serve as a starting point for your DLA program. Modify your program to perform the DLA process on a 101×101 lattice—we choose an odd length for the side of the square so that there is one lattice site exactly in the center. Repeatedly introduce a new particle at the center and have it walk randomly until it sticks to an edge or an anchored particle.

   You will need to decide some things. How are you going to store the positions of the anchored particles? On each step of the random walk you will have to check the particle's neighboring squares to see if they are outside the edge of the system or are occupied by an anchored particle. How are you going to do this? You should also modify your visualization code from the Brownian motion exercise to visualize the positions of both the randomly walking particles and the anchored particles. Run your program for a while and observe what it does.

b) In the interests of speed, change your program so that it shows only the anchored particles on the screen and not the randomly walking ones. That way you need update the pictures on the screen only when a new particle becomes anchored. Also remove any `rate` statements that you added to make the animation smooth.

   Set up the program so that it stops running once there is an anchored particle in the center of the grid, at the point where each particle starts its random walk. Once there is a particle at this point, there's no point running any longer because any further particles added will be anchored the moment they start out.

Run your program and see what it produces. If you are feeling patient, try modifying it to use a 201×201 lattice and run it again—the pictures will be more impressive, but you'll have to wait longer to generate them.

A nice further twist is to modify the program so that the anchored particles are shown in different shades or colors depending on their age, with the shades or colors changing gradually from the first particle added to the last.

c) If you are feeling particularly ambitious, try the following. The original version of DLA was a bit different from the version above—and more difficult to do. In the original version you start off with a single *anchored* particle at the center of the grid and a new particle starts from a random point on the perimeter and walks until it sticks to the particle in the middle. Then the next particle starts from the perimeter and walks until it sticks to one of the other two, and so on. Particles no longer stick to the walls, but they are not allowed to walk off the edge of the grid.

Unfortunately, simulating this version of DLA directly takes forever—the single anchored particle in the middle of the grid is difficult for a random walker to find, so you have to wait a long time even for just one particle to finish its random walk. But you can speed it up using a clever trick: when the randomly walking particle does finally find its way to the center, it will cross any circle around the center at a random point—no point on the circle is special so the particle will just cross anywhere. But in that case we need not wait the long time required for the particle to make its way to the center and cross that circle. We can just cut to the chase and start the particle on the circle at a random point, rather than at the boundary of the grid. Thus the procedure for simulating this version of DLA is as follows:

i) Start with a single anchored particle in the middle of the grid. Define a variable r to record the furthest distance of any anchored particle from the center of the grid. Initially $r = 0$.
ii) For each additional particle, start the particle at a random point around a circle centered on the center of the grid and having radius $r + 1$. You may not be able to start exactly on the circle, if the chosen random point doesn't fall precisely on a grid point, in which case start on the nearest grid point outside the circle.
iii) Perform a random walk until the particle sticks to another one, except that if the particle ever gets more than $2r$ away from the center, throw it away and start a new particle at a random point on the circle again.
iv) Every time a particle sticks, calculate its distance from the center and if that distance is greater than the current value of r, update r to the new value.
v) The program stops running once r surpasses a half of the distance from the center of the grid to the boundary, to prevent particles from ever walking outside the grid.

Try running your program with a 101×101 grid initially and see what you get.