

Exercise 1.1: 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

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

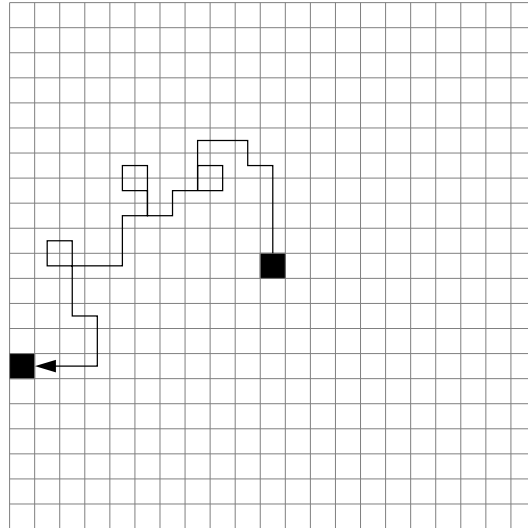
where the notation $\langle ij \rangle$ indicates a sum over pairs i, j that are adjacent on the lattice. On the square lattice we use in this exercise each spin has four adjacent neighbors with which it interacts.

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 $-J s_1 s_2$ if spins 1 and 2 are adjacent to one another, but you do not also need a term $-J s_2 s_1$. 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, say, 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. 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, 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 1.2: Diffusion-limited aggregation

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) Modify your random walk 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 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.

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.

Exercise 1.3: Vibration in a one-dimensional system

In Advanced Classical Mechanics we studied the motion of a system of N identical masses (in zero gravity) joined by identical linear springs like this:



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 . We know 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 ζ_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.

Exercise 1.4: Quantum oscillators

Consider the one-dimensional, time-independent Schrödinger equation in a harmonic (i.e., quadratic) potential $V(x) = V_0 x^2 / a^2$, 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. Write a program, to find the energies of the ground state and the first two excited states for these equations when m is the electron mass, $V_0 = 50$ eV, and $a = 10^{-11}$ 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.)

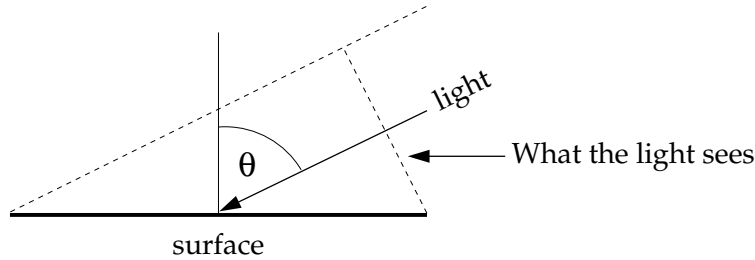
- b) Now modify your program to calculate the same three energies for the anharmonic oscillator with $V(x) = V_0 x^4 / a^4$, with the same parameter values.
- c) Modify your program further to calculate the properly normalized wavefunctions of the anharmonic oscillator for the three states and make a 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 1.5: 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.

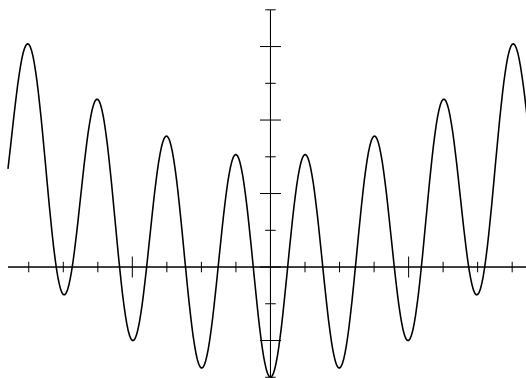
- A file called `altitude.txt` is attached, which contains the altitude $w(x, y)$ in meters above sea level (or depth below sea level) of the surface of the Earth, measured on a grid of points (x, y) . Write a program that reads this file and stores the data in an array. Then calculate the derivatives $\partial w/\partial x$ 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.)
- Now, using your values for the derivatives, calculate the intensity for each grid point, with $\phi = 45^\circ$, 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) Another file called `stm.txt` is attached, 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).

Exercise 1.6: Global minimum of a function

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



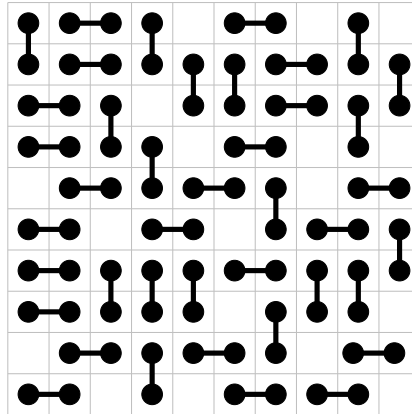
Clearly the global minimum of this function is at $x = 0$.

- Write a program to confirm this fact using simulated annealing starting at, say, $x = 2$, with Monte Carlo moves of the form $x \rightarrow x + \delta$ 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.
- 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 1.7: 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.)

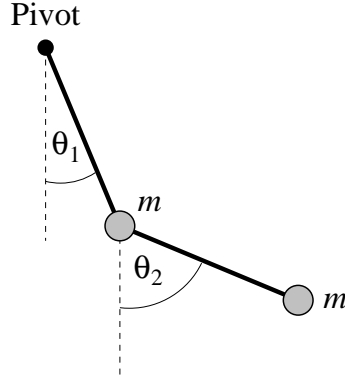
- a) Write a program to solve the problem using simulated annealing on a 50×50 lattice. The “energy” function for the system is *minus* the number of dimers, so that it is minimized when the dimers are a maximum. The moves for the Markov chain are as follows:
 - i) Choose two adjacent sites on the lattice 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 1.8: 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

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

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

$$v_1 = \ell \dot{\theta}_1, \quad v_2^2 = \ell^2 [\dot{\theta}_1^2 + \dot{\theta}_2^2 + 2\dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2)],$$

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

$$T = \frac{1}{2}mv_1^2 + \frac{1}{2}mv_2^2 = m\ell^2 [\dot{\theta}_1^2 + \frac{1}{2}\dot{\theta}_2^2 + \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2)],$$

and the Lagrangian of the system is

$$\mathcal{L} = T - V = m\ell^2 [\dot{\theta}_1^2 + \frac{1}{2}\dot{\theta}_2^2 + \dot{\theta}_1 \dot{\theta}_2 \cos(\theta_1 - \theta_2)] + mg\ell(2 \cos \theta_1 + \cos \theta_2).$$

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

$$\frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{\theta}_1} \right) = \frac{\partial \mathcal{L}}{\partial \theta_1}, \quad \frac{d}{dt} \left(\frac{\partial \mathcal{L}}{\partial \dot{\theta}_2} \right) = \frac{\partial \mathcal{L}}{\partial \theta_2},$$

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:

$$\dot{\theta}_1 = \omega_1, \quad \dot{\theta}_2 = \omega_2.$$

In terms of these variables our equations of motion become

$$2\dot{\omega}_1 + \dot{\omega}_2 \cos(\theta_1 - \theta_2) + \omega_2^2 \sin(\theta_1 - \theta_2) + 2\frac{g}{\ell} \sin \theta_1 = 0,$$

$$\dot{\omega}_2 + \dot{\omega}_1 \cos(\theta_1 - \theta_2) - \omega_1^2 \sin(\theta_1 - \theta_2) + \frac{g}{\ell} \sin \theta_2 = 0.$$

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

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

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

(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.

- 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 .
- Write a program using the fourth-order Runge–Kutta method to solve the equations of motion for the case where $\ell = 40$ cm, with the initial conditions $\theta_1 = \theta_2 = 90^\circ$ 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.

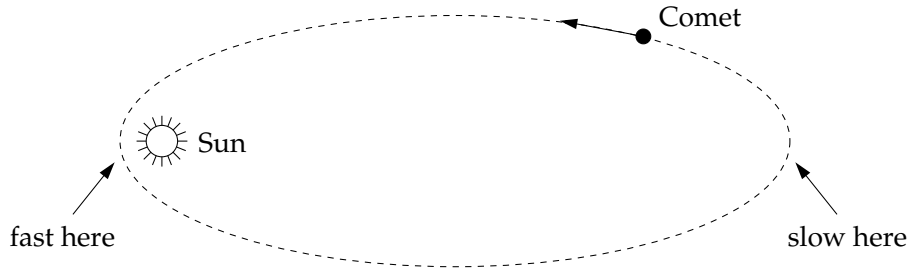
Because of energy conservation, the total energy should be constant over time (actually it should be zero for this particular set of initial conditions), but you will find that it is not perfectly constant because of the approximate nature of the solution of the differential equation. Choose a suitable 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.

- 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 1.9: 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

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

Canceling the m and taking the x component we have

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

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:

$$\frac{d^2 x}{dt^2} = -GM \frac{x}{r^3}, \quad \frac{d^2 y}{dt^2} = -GM \frac{y}{r^3},$$

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

- Turn these two second-order equations into four first-order equations, using the methods you have learned.
- 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 $v_y = 500 \text{ m s}^{-1}$. 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?

- 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 1.10: The three-body problem

If you mastered the problem 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.

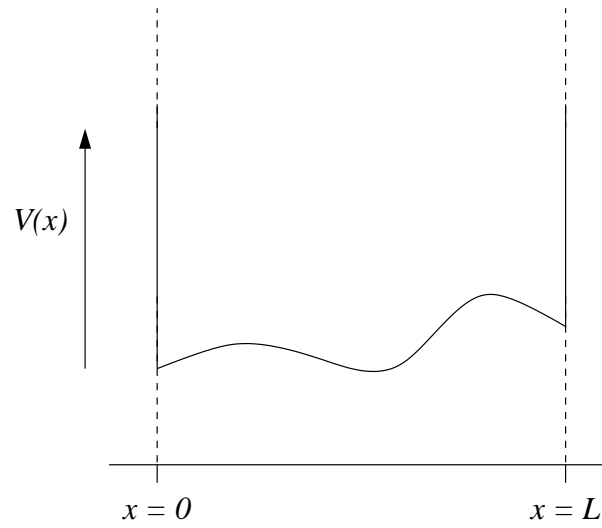
To do this calculation properly you will need to use an adaptive step size method, for the same reasons as in Exercise 8.10—the stars move very rapidly when they are close together and very slowly when they are far apart. An adaptive method is the only way to get the accuracy you need in the fast-moving 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 1.11: 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

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

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 ψ_1, ψ_2, \dots are the Fourier coefficients.

a) Noting that, for m, n positive integers

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

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

$$\sum_{n=1}^{\infty} \psi_n \int_0^L \sin \frac{\pi m x}{L} \hat{H} \sin \frac{\pi n x}{L} dx = \frac{1}{2} L E \psi_m.$$

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

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

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 (ψ_1, ψ_2, \dots) . 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

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

Write a 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 \AA , and $a = 10 \text{ eV}$. (The mass and charge of an electron are $9.1094 \times 10^{-31} \text{ kg}$ and $1.6022 \times 10^{-19} \text{ C}$ 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 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 .
- 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 $\int_0^L |\psi(x)|^2 dx = 1$. Is this true of your wavefunction?

Exercise 1.12: 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

$$b(x, y) = \int_0^K \int_0^L a(x', y') f(x - x', y - y') dx' dy',$$

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

$$b(x) = \int_0^L a(x') f(x - x') dx'.$$

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

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

where

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

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

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

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

$$\tilde{b}_k = \frac{1}{L} \int_0^L a(x') \exp\left(-i \frac{2\pi k x'}{L}\right) \int_{-x'}^{L-x'} f(X) \exp\left(-i \frac{2\pi k X}{L}\right) dX dx'.$$

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

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

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

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

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 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.

Exercise 1.13: Driven Damped Oscillator and Its Application in Non-Linear Dynamics

Exercise 1.14: Solution of Schrödinger equation