Computational Physics

Practical exercises

David Buscher

Version 4.0 February 2024

Contents

Introduction

These exercises are designed to help you learn some `Python`, understand some physics and learn a little more about numerical programming. You should attempt the exercises in the order given. Demonstrators will be on hand during the practical sessions to assist.

The speed at which people can program varies widely. It even varies for individuals day by day — all programmers will sympathise with how much time can be wasted trying to fix an obscure bug. So do

keep an eye on the clock, and do seek assistance. Talk to your colleagues so you can share experience and spot each others' mistakes. The goal of the exercises is for you to learn – and you can only do this by experience.

Each exercise is split into tasks. The *core task or tasks* are ones that I hope you will all be able to achieve in the class in the time available. If you hand in a working solution to this you will get approximately 60–70% of the credit for the exercise. There is also one or more *supplementary tasks* per exercise, which take the problem further, or in a different direction. I hope that many of you will achieve some or most of these, and they count for the remaining fraction of the credit.

I have given structured hints to each problem. In addition, you should look at the code examples in the lectures and, importantly, **consult the relevant documentation for the libraries** such as `scipy` and `numpy` — finding and learning from online documentation and code examples is an important skill. There are some starting links from the course web-page or just use Google/Bing/DuckDuckGo/Kagi.

Assessment

Over the 4 weeks you should carry out three exercises. The marks available are a maximum of 6 for exercises 1 and 2, and a maximum of 8, for a total of 20 marks.

You should upload your solution to the TiS. The **deadline** is posted on the course web site and the TiS. You are encouraged to submit your work as soon as it is in a reasonable state, and move on. So don't spend too much time on the tasks, but don't spend too little time either.

For each exercise you should submit a Jupyter notebook file (a `.ipynb` file) which is structured into tasks as explained in the next section. This notebook will contain Python code, plots illustrating relevant results, and text commentary. The text commentary describes very briefly what you did, any major problems, mentions any numerical answers required and describes the accompanying plots.

It is possible in Jupyter notebooks to execute your code cells in any order. You should make sure that, in your submitted notebook, the code will run correctly when you restart the Python interpreter and run all the cells in the order that they appear (for example using the "Resart and run all" function under the "Runtime" menu in Google Colab). This is so that the logical order that the code should be run is clear to someone reading the code. Breaking your code up into functions will help with the organisation and ordering of your code especially when you need to do different variations of the same calculation.

Note that the usual rules with respect to plagiarism apply: you are of course allowed to discuss the work with others and to refer to online or other sources, but you must clearly state where any of the work you submit is not your own work, or is joint work, and explicitly attribute the authors/sources of that work.

Code quality

In the first two exercises, the main emphasis of the marking will be on successful completion of the tasks. In the last exercise a percentage of the marks will be given for code quality (this will also be true for the optional project). High-quality code will include at minimum:

- Structured code: tasks broken down into sensible functions;
- Meaningful function names;
- Meaningful variable names (this is less important than for naming functions: single-letter variable names can be meaningful if the meaning can be inferred from context, e.g. loop counters);
- Appropriate levels of commenting (at minimum a Python "docstring" identifying what each function does);
- Sensible use of whitespace to indicate code structure.

Remember, code quality is important because (a) it helps to make the code easier to understand and debug for the person writing the code and (b) it makes the code easier to understand and debug for others who may have to modify the program later on.

Writing high-quality code is somewhat like writing high-quality prose or mathematics: clearly structuring and explaining your thoughts for someone else can help clarify your own thinking, and this clear explanation is what code quality is all about.

Getting started

Using Google Colab

We will be using the Jupyter notebooks for these exercises. You can run Jupyter on your own computer or use the notebooks provided by Google Colab at https://colab.research.google.com. The following assumes that you will be using Google Colab, but most of the advice should apply to any Jupyter notebook or Jupyter lab installation.

It is best to log in with your University of Cambridge account: you can do this by logging out of any other Google accounts you may be logged into and then logging back in using `crsid@cam.ac.uk` as your username — if you are then prompted you about a choice of accounts associated with this email address, always choose the "Google workspace account" rather than the "Personal account". You can also use Colab with any other Google account you may have, but this may give rise to problems when you want to get access to any shared files or to share notebooks with others, e.g. demonstrators.

There are plenty of tutorials available in Google Colab and elsewhere about how to use Jupyter notebooks, but an effective way to learn for many people is to just open a new notebook and start experimenting. To do this, click on the "New notebook" link in Colab. The first thing you should do after

creating a new notebook is to rename the notebook by clicking on the existing title (usually something like “Untitled0.ipynb”) and editing it to give it a descriptive name such as “Exercise 1”.

Structuring the notebook

One feature of Jupyter notebooks we will be making use of in the exercises is the use of text cells as well as code cells. Code cells are the default cell type and contain the Python code you are developing. You will use text cells to put in section headings to separate different tasks, and also to put in text commentary and conclusions.

The text cells use a format called “markdown” which allows you to type in plain text into the cell and get various “rich text” effects such as section headings, bold, italic, and even nicely-formatted mathematics when the cell is “run”. You can find out more about markdown online, for example at https://colab.research.google.com/notebooks/markdown_guide.ipynb.

With this in mind, you should insert a text cell (you can use the + `Text` menu item near the top of the screen for this) at the beginning of your notebook that gives it a section heading, for example:

```
# Core task 1

Some text describing the task (optional - this can be used to
remind yourself what you are aiming to do).
```

When you press the arrow to “run” the cell it will turn it into a bold-font first-level heading, plus any text you typed in.

After that you should probably insert your first code cell. A suggested first cell is one which just imports all the relevant modules, e.g.

```
import matplotlib.pyplot as plt
import numpy as np
import scipy
```

You can then run this cell and in the next code cell start writing the first part of your code.

At the end of each task, you should add a text cell with a conclusions sub-heading and text in the form

```
## Conclusions

Some brief words here about what you did and what
you achieved in this task.
Any numerical results should be stated here.
```

You can of course add additional text cells in the notebook to annotate your work for yourself and/or the assessors. This section structure should be repeated for all the core and supplementary tasks.

You should submit one notebook per exercise, containing the code and results for all the tasks in that exercise.

Plotting your results

You will want to present many of your results in the form of graphs made using pyplot, which will appear below the code cell when the cell is run. **Remember to add captions for the axes and a title.** Here is a simple Python code fragment which plots a sine function with axes.

```
plt.plot(np.sin(np.linspace(-10,10,100)))
plt.xlabel("Distance along axis (m)")
plt.ylabel("Intensity (arbitrary units)")
plt.title("Diffraction pattern of a double slit");
```

Note the use of a semicolon on the last line to suppress any printed output — not essential but makes the notebook look cleaner. You can also use the "object-oriented" pyplot plotting style used in places in the lectures, which can be helpful when you are plotting multiple plots in the same figure.

It is usually a good idea to do the long-running part of any calculation in one code cell and then do the plotting in a subsequent code cell. This way, you can change the plotting code and re-run it to improve the appearance of the plot and add labels etc without re-running all of the calculations.

If you are doing very-long-running calculations you may want to save the results to a file to analyse/-plot later. You can use `numpy.savez()` function to do this.

Uploading your notebook to the TiS

You should use one notebook per exercise. When you are finished the exercise and are ready to submit it for assessment, you can use the "Download .ipynb" item under the "File" menu in Colab. You can then upload the file from your computer to the TiS.

Exercise 1: The Driven Pendulum

Goal

To explore the physics of a non-linear oscillator (a damped, driven pendulum) by accurate integration of its equation of motion.

Physics

The pendulum comprises a bob of mass m on a light rod of length l and swings in a uniform gravitational field g. If there is a resistive force equal to αv where the bob speed is v, and a driving sinusoidal couple G at frequency Ω_D, we can write

$$m\,l^2\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -m\,g\,l\,\sin(\theta) - \alpha\,l\,\frac{\mathrm{d}\theta}{\mathrm{d}t} + G\sin(\Omega_D t) \tag{1}$$

Rearranging:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{l}\,\sin(\theta) - q\,\frac{\mathrm{d}\theta}{\mathrm{d}t} + F\sin(\Omega_D t) \tag{2}$$

where we have defined $q \equiv \alpha/(m\,l)$ and $F \equiv G/(m\,l^2)$.

For the purposes of this exercise, take $l = g$, so the natural period for small oscillations should be 2π seconds. Also let the driving angular frequency be $\Omega = 2/3\mathrm{rad\,s^{-1}}$. This leaves q and F, and the initial conditions, to be varied. For all of these problems, we will start the pendulum from rest i.e. $\dot\theta = 0$ at $t = 0$, but we will vary the initial displacement θ_0. The parameter space to explore then is in the three values q, F and θ_0.

Tasks

Core Task 1 First re-write this second-order differential equation Equation 2 as a pair of linked first-order equations in the variables $y_0 = \theta$ and $y_1 = \omega = \dot\theta$. Now write a program that will integrate this pair of equations using a suitable algorithm, from a given starting point $\theta = \theta_0$ and $\omega = \omega_0$ at $t = 0$. I recommend using `scipy` rather than implementing your own Runge-Kutta or other technique.

Test the code by setting $q = F = 0$ and starting from $(\theta_0, \omega_0) = (0.01, 0.0)$, and plotting the solution for 10, 100, 1000…natural periods of oscillation. Overlay on your plot the expected theoretical result for small-angle oscillations — make sure they agree!

To Do: Refer to hint 4.

Test how well your integrator conserves energy: run the code for, say, 10,000 oscillations and plot the evolution of energy with time.

Now find how the amplitude of undriven, undamped oscillations depends affects the period. Plot a graph of the period T versus θ_0 for $0 < \theta_0 < \pi$.

== initial angle theta_0

> **Include in your notebook:** Source code, appropriately-scaled plots showing how well energy is conserved in at least one case and a plot of period versus amplitude, conclusions text containing a couple of sentences summarising what you managed to achieve, and the value of the period for $\theta_0 = \pi/2$

Core task 2 Now turn on some damping, say $q = 1, 5, 10$, plot some results, and check that the results make sense. Now turn on the driving force, leaving $q = 0.5$ from now on, and investigate with suitable plots the behaviour for $F = 0.5, 1.2, 1.44, 1.465$. What happens to the period of oscillation? *Note that the period of oscillation is best observed in the angular velocity rather than angular position, to avoid problems with wrap-around at $\pm\pi$.*

> **Include in your notebook:** Source code, a sentence or two in the conclusions sub-section explaining what you see in the solutions, and illustrative plots of the displacement θ and the angular velocity $\frac{\mathrm{d}\theta}{\mathrm{d}t}$ versus time.

Supplementary task 1 Investigate the sensitivity to initial conditions: compare two oscillations, one with $F = 1.2, \theta_0 = 0.2$ and one with $F = 1.2, \theta_0 = 0.20001$. Integrate for a 'long time' to see if the solutions diverge or stay the same.

> **Include in your notebook:** Source code, a sentence or two in the conclusions sub-section explaining what you see in the solutions, and illustrative plots of the behaviour.

Supplementary Task 2 Try plotting angle versus angular speed for various solutions, to compare the type of behaviour in various regimes: you can investigate chaotic behaviour using this simple code — have a look in the books or a web site for examples. There is a nice demo for example at http://www.myphysicslab.com. There's lots more physics to be explored here — experiment if you have time!

> **Include in your notebook:** Source code, a sentence or two in the conclusions sub-section explaining what you see in the solutions, and illustrative plots of the behaviour.

Hints

1. Hopefully rewriting the equation as 2 ODEs is straightforward: if not, ask!
2. I recommend that you use a "canned" ODE integrator, for example the `scipy.integrate.solve_ivp()` function, rather than writing your own.
3. You will need to write a function that evaluates the derivatives of the ODEs at a given time given the current values of the variables. You can look at the example code solving the spinning ring problem discussed in lectures. You can perhaps adapt some of that code if you get stuck.
4. Think about the time window you choose for your plots: in some cases, plotting less than the full set of data you have computed may make for clearer visualisation of what is going on.

It means time slices

5. Many of the ODE integrator functions in `scipy.integrate` and elsewhere are adaptive-stepping algorithms, and have options which are set using "keyword arguments" to control this

behaviour. Firstly, you can set a desired accuracy for the solution: setting a higher accuracy may make the program run more slowly because it will typically take shorter steps. Secondly, the functions usually have the capability to return an interpolated value of the solution at user-specified times, and not just at the time steps used to integrate the ODE. You can choose a time sampling to give appropriately smooth plots. It pays to experiment with these values to see if they have any effect on your results, especially when investigating "chaotic" behaviour.

6. To find the period versus amplitude relationship, a simple (and just about acceptable) way is to measure the period off a suitable plot, and do this for several values of θ_0. However, it is much better to alter your code to estimate the period directly. You can then loop over θ_0 values and plot the period versus amplitude relation. Two obvious approaches spring to mind. First, you can find when θ first goes negative. This is when the time is approximately $T/4$ where T is the period. How accurate would this result be? You could also find several zero crossings by considering when y changes sign or becomes exactly zero after a step is taken; by counting many such zero crossings and recording the time between them you can get a more accurate value for T.
7. Note that you need to think about what happens when the pendulum goes "over the top" and comes down the other side — you need to think about the 2π ambiguities involved, and what this means in terms of the "period" of an oscillation.

Exercise 2: Fraunhofer and Fresnel Diffraction

Goal

Write a program to calculate the near and far-field diffraction patterns of an arbitrary one-dimensional complex aperture using the Fast Fourier Transform technique. Test this program by using simple test apertures (a slit) for which the theoretical pattern is known. Investigate more complicated apertures for which analytical results are difficult to compute.

Physics

Plane monochromatic waves, of wavelength λ, arrive at normal incidence on an aperture, which has a complex transmittance $A(x)$. The wave is diffracted, and the pattern is observed on a screen a distance D from the aperture and parallel to it. We want to calculate the pattern when the screen is in the far-field of the aperture (Fraunhofer diffraction) and also in the near-field (Fresnel).

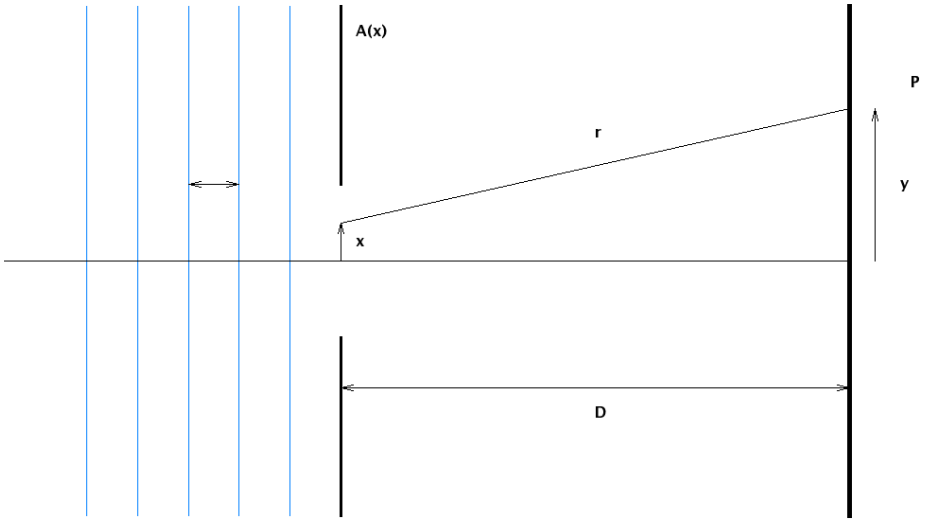

Figure 1: Geometry for diffraction calculation

Using Huygen's construction, we can write the disturbance at a point P on the screen, a distance y from the axis, as

$$\psi(y) \propto \int \frac{A(x)\, \exp(ikr)}{r}\, \mathrm{d}x$$

where $k = 2\pi/\lambda$. We have assumed that all angles are small:

$$x, y \ll D$$

so that we are close to the straight-through axis and can therefore neglect terms like $\cos(\theta)$ which appear if we are off-axis. We now expand the path length r in powers of x/r:

$$r^2 = D^2 + (y-x)^2$$

$$r \approx D + \frac{y^2}{2D} - \frac{xy}{D} + \frac{x^2}{2D} + \mathcal{O}\left(\frac{(y-x)^4}{D^3}\right)$$

If we now neglect the variation in r in the denominator of the integral, setting $r \approx D$, which is adequate for $x, y \ll D$, then we can write

$$\psi(y) \propto \frac{\exp(ikD)}{D}\, \exp\left(\frac{iky^2}{2D}\right) \int A(x)\, \exp\left(\frac{ikx^2}{2D}\right) \exp\left(\frac{-ikxy}{D}\right)\, \mathrm{d}x \tag{3}$$

The diffraction pattern is thus the Fourier-transform of the modified aperture function A':

$$\psi(y) \propto \exp\left(\frac{iky^2}{2D}\right) \int A'(x)\ \exp\left(\frac{-ikxy}{D}\right) \mathrm{d}x \tag{4}$$

with

$$A'(x) = \exp\left(\frac{ikx^2}{2D}\right) A(x) \tag{5}$$

In the far-field (Franhofer limit) we have $kx^2/(2D) \ll \pi$ so that $A' \approx A$ for all values of x in the aperture where $A(x)$ is non-zero, i.e. the familiar result

see Franhofer condition

same

see Franhofer condition

$$d \gg \frac{x_{\text{max}}^2}{\lambda}.$$

The distance x_{max}^2/λ is the Fresnel distance. In this case, the diffraction pattern is just the Fourier transform of the aperture function.

Note that we can calculate the near-field (Fresnel) pattern also if we include a step to modify the aperture function according to Equation 5 *before we take its Fourier transform*.

Note that if we are only interested in the pattern's intensity, we can ignore the phase prefactor in Equation 4.

Finally, we can discretize Equation 4, by sampling the aperture evenly at positions x_j

$$\psi(y) \propto \Delta \sum_{j=0}^{N-1} A'(x_j)\ \exp\left(\frac{-ikx_jy}{D}\right) \tag{6}$$

where Δ is the distance between the aperture sample positions x_j.

One convenient definition of the sample points in x is

$$x_j = (j - (N/2))\Delta, \tag{7}$$

where N is the number of sample points in the aperture. Note that this definition of the x-coordinate is equivalent to applying a coordinate transform equivalent to the "fftshift" operation described in the lectures, and results in Fourier phases which are closer to zero compared to a more simple linear relationship between x_j and j.

Tasks

Core Task 1 : Write a program that will calculate the diffraction pattern of a general 1-dimensional complex aperture in the far field of the aperture using FFT techniques. The program should calculate

the intensity in the pattern across the screen, which you should plot using the correct y coordinates (in metres or microns for example). **Label your coordinates.**

Test this program for the specific case of a slit in the centre of an otherwise blocked aperture: take the single slit to have width d in the centre of an aperture of total extent L. For definiteness, use $\lambda = 500$ nm, $d = 100\,\mu$m, $D = 1.0$ m and $L = 5$ mm. Overlay on your plot the theoretical value of the intensity pattern expected.

Core Task 2 : Now calculate and plot the Fraunhofer diffraction pattern of a *sinusoidal phase grating*. This grating is a slit of extent $d = 2$mm, outside of which the transmission is zero. Within $|x| < d/2$, the transmission amplitude is 1.0, and the phase of A is

$$\phi(x) = (m/2)\sin(2\pi x/s)$$

where s is the spacing of the phase maxima, and can be taken as 100microns for this problem. For this calculation, use $m = 8$. The Fresnel distance d^2/λ is 8 m, so calculate the pattern on a screen at $D = 10$ m. What do you notice about the resulting pattern?

Core task 3 : Now modify your program so that the calculation is accurate even in the near-field by adding a phase correction to the aperture function as defined by Equation 5. Repeat your calculations in the previous two tasks for $D = 5$ mm for the slit, and $D = 0.5$ m for the phase grating, and plot the results. Do the intensity patterns look sensible?

Supplementary Task 1 : Write a program to evaluate the Fresnel integrals *accurately* using a standard integration routine from the `scipy`, and use this to make a plot of the Cornu spiral using `pyplot`. Do not use a Monte-Carlo routine — they are not efficient for low-dimensional integrals — instead use a standard quadrature technique. One version of the Fresnel integrals can be written

$$C(u) = \int_0^u \cos\left(\frac{\pi\, x^2}{2}\right)\, dx$$

$$S(u) = \int_0^u \sin\left(\frac{\pi\, x^2}{2}\right)\, dx$$

Note that there is a function `scipy.special.fresnel()` whose only purpose is to evaluate this integral! However using this special-purpose function to compute your answer negates the point of understanding how to use a general-purpose integrator and will not be looked on favourably.

Supplementary Task 2 : Use the Fresnel integrals computed in Supplementary Task 1 to re-calculate the diffraction pattern of the near field slit in Core Task 3 (not the phase grating). Physics reminder: the complex amplitude in the near field on the axis of an open slit is given by

$$\Psi \propto \int_{x_0}^{x1} \cos(\pi x^2/(\lambda D)) + i \sin(\pi x^2/(\lambda D))\, dx,$$

so you can use the Fresnel integrals directly by scaling the length dimensions by $\sqrt{2/(\lambda D)}$. To calculate the intensity pattern as we move off-axis, we can imagine translating the slit relative to the screen and change the integration limits appropriately. Compare the results for this task with the results from the Fourier method in Core Task 3.

Hints

Diffraction calculations using the FFT Recall the DFT definition:

$$H_j = \sum_{m=0}^{N-1} h_m\, e^{2\pi\, i\, m\, j/N} \tag{8}$$

which maps N time-domain samples h_m into N frequencies, which are

$$f_j = \frac{j}{N\Delta} \tag{9}$$

You can think of frequencies $(j/N) \times (1/\Delta)$, running from $j = 0$ to $(N - 1)$, with

- $j = 0$ is zero frequency.
- For $1 \leq n \leq (N/2)$, we have positive frequencies $(j/N) \times (1/\Delta)$.
- For $(N/2) + 1 \leq j \leq (N - 1)$ we have *negative frequencies* which we compute as $((j/N) - 1) \times (1/\Delta)$. (Remember the sequences are periodic).

Of course in this case we don't have time-domain samples, but we can still use the FFT to carry out the transform.

The complex aperture function will be represented by an array of N discrete complex values along the aperture, encoding the real and imaginary parts of $A(x)$. Each complex value represents the aperture's transmittance over a small length Δ of the aperture, so that $N\Delta$ is the total extent of the aperture.

Choose appropriate values for N and Δ to make sure you can represent the whole aperture of maximum extent L well enough. Bear in mind that Fast Fourier Transform calculations are fastest when the transform length is a power of 2, and that you want Δ to be small enough to resolve the features

of the aperture. (Computers are fast these days though; so you can use small values of Δ and correspondingly large values of N. In practice, for such a small problem, the use of N as a power of 2 is not necessary, but it is important if performance is critical.)

For the slit problem you can use the `numpy.zeros()` function to set up an array of the appropriate size filled with zeros and then set the locations where the slit is transparent by assigning non-zero values to "slices" e.g. `a[5:10]=1.0`.

You now need a routine to calculate the fast Fourier transform (FFT). You can use the `numpy.fft.fft()` function, which is straightforward to use (see the code examples from the lectures). It accepts a real or a complex input and produces a complex output. You will need to compute the intensity of the pattern. You might also want to plot the aperture function to make sure you have calculated it correctly.

Now think carefully about the coordinates associated with your calculated pattern. The discussion at the beginning of this section reminds you about how the frequencies appear in the FFT'ed data. Can you understand the form of the intensity pattern you have derived?

To plot the intensity on the screen as a function of actual distance y, you need to work out how to convert the pixels in the Fourier transform into distances on the screen y. To do this you first need to compare carefully Equation 6 and Equation 8 (also referring to Equation 7) which should tell you how to derive y at each pixel value. In addition, by interpreting the second half of the transform as negative frequencies (or y values in this case) you should be able to plot the intensity pattern as a function of y for positive and negative y, and plot over this the matching sinc function for a slit. **If you are having difficulties with this step it may help to revise your notes from the lectures concerning the location of negative frequencies in the output of an FFT**.

Numerical Integration with `scipy` You will need to choose a `scipy` routine to do the integration. The `scipy.integrate.quad()` function is a good general purpose integrator. Example code is available online: search for "scipy numerical integration examples".

Write suitable functions that evaluate the two integrands. Evaluate the integrals for various values of s and use `pyplot` to plot the spiral.

Plotting the diffraction pattern should now be straightforward. Remember to think carefully about the coordinates associated with your calculated pattern.

Exercise 3: The 2D Ising model

Goals

The Ising model is a well-known way of describing ferromagnetism using simplified spin-spin interactions. We will construct a 2D Ising model and use it to measure the thermodynamic and magnetic properties of the system as a function of time and as a function of the temperature and magnetic field.

Physics

The 2D version of the Ising model considers an w by w square lattice, resulting in $N = w^2$ spin sites. It is conventional to use *periodic boundary conditions*, where the lattice "wraps round" from the left edge to the right edge and from the top edge to the bottom edge; topologically the system is then the same as the surface of a toroid. With these boundary conditions every spin has 4 neighbours. The energy of the system is then given by

$$E = -J \sum_i \sum_j s_i s_j - \mu H \sum_{i=1}^{N} s_i$$

where we sum over all spins s_i and their neighbours s_j with exchange energy J, magnetic moment μ, and externally applied magnetic field H.

The most interesting variable of the system is the relative magnetisation of the complete lattice

$$M = \frac{1}{N} \sum_{i}^{N} s_i$$

or the expected total magnetisation

$$\mathbb{E}[M(\mathbf{s})] = \sum_{s \in \{-1,1\}^N} M(s) p_{\mathbf{s}}(s)$$

with $p_{\mathbf{s}}(s)$ the probability mass function. The magnetisation shows a very strong dependence on the temperature as well as the externally applied magnetic field H. The magnetisation can also show a high variance.

The analytical approach to calculate these expected values would be to use the probability mass function

$$p_{\mathbf{s}}(s) = \frac{1}{Z} e^{-\beta E}$$

with partition function

$$Z = \sum_{s \in \{-1,1\}^N} e^{-\beta E}$$

and inverse temperature $\beta = \frac{1}{k_b T}$. However, computing this type of partition function quickly becomes arduous or simply infeasible for a lattice with any interesting number of spins, especially if the variance is also high. We thus typically use Monte Carlo methods to investigate such systems and estimate the magnetisation M.

The Metropolis-Hastings algorithm The basic algorithm is as follows: we construct a w by w grid of spins, starting from some initial set of spin orientations, perhaps a random or uniform set, the model is evolved in time by a Monte-Carlo technique: pick a spin, and calculate the energy ΔE required to flip it. If $\Delta E < 0$, flip the spin; if $\exp(-\Delta E/(k_B T)) > p$, where p is a random number drawn from a uniform distribution in the range $[0, 1]$, then flip the spin; else do nothing.

We now repeat this step for many lattice sites. One can either step systematically through the lattice doing N possible flips, or pick N random points. You can think of these N operations together as a single *time step*. We repeat many steps to evolve the spin system. We wait for the system to reach equilibrium and then sample relevant quantities and average them over a large number of time steps.

Tasks

Core Task 1: No spin coupling Write a program to construct a lattice and evolve it with time according to the Metropolis-Hastings algorithm. It is probably simplest to parameterise your functions in terms of the dimensionless quantities βJ and $\beta \mu H$.

Set $\beta J = 0$ so there is no coupling. In such a case, only thermal motion and the external magnetic field affect the flipping of the spins. Do a few simulations with $-3 < \beta \mu H < 3$ and observe what happens by plotting the lattice configuration after 1, 10, 100, 1000, 10,000 and (if you have time) 100,000 "time steps" (you can display the lattice configuration using the `matplotlib.pyplot.matshow()` function). Try both starting with a random lattice and a lattice with all spins in one direction.

Compute the relative magnetisation M of the lattice and plot it as a function of time for at least 10,000 steps. Qualitatively explain the system behaviour.

In the case of no coupling, an analytical expression for the mean magnetisation $\langle M \rangle$ exists. The expression is:

$$\langle M \rangle = \tanh(\beta \mu H)$$

Test whether the Monte Carlo method correctly reproduces the above analytical formula.

Core Task 2: only spin coupling Now let us investigate coupling without an external magnetic field ($H = 0$). Try the same as previously, using different starting conditions, but introduce some ferromagnetic coupling ($\beta J = 0.2$) or anti-ferromagnetic coupling ($\beta J = -0.2$). Qualitatively explain the system behaviour.

For this case, no field and only coupling, Lars Onsanger discovered an analytical solution for the case of an infinite lattice:

$$\langle M \rangle = \begin{cases} 0 & \text{if } T \geq T_c \\ \pm \left(1 - \sinh\left(2\beta J\right)^{-4}\right)^{\frac{1}{8}} & \text{if } T < T_c \end{cases}$$

where T_c is the critical temperature given by

$$T_c = \frac{2J}{k_b \ln(1+\sqrt{2})}.$$

Test whether the Monte Carlo method correctly reproduces the above analytical formula. Comment on any unusual behaviour of the system you notice around the critical point.

Hint: The system will equilibrate much faster if you start with an initial state that is close to the expected equilibrium state. This can be achieved, for example, by using the final lattice state from a simulation at one temperature as the starting point of a new simulation at a nearby temperature.

Supplementary task: Investigate the heat capacity near the phase transition. The fluctuation-dissipation theorem states that the heat capacity of the system is given by

$$C = \frac{\sigma_E}{k_B T^2}$$

with σ_E being the standard deviation of fluctuations in the system energy E. Plot a graph of the heat capacity of the system as a function of temperature for $H = 0$, evaluating what happens close to the critical temperature T_c for a number of different lattice sizes.

Hints

Results get more representative of a real ferromagnetic system as we go to larger and larger lattice sizes w, but larger lattices take more computing time, so make sure you write functions which can generate and evolve an arbitrary-sized (but square) lattice, and then experiment with different-sized lattices to see how large a lattice is practical.

At the same time, to get accurate values for mean quantities such as the magnetisation requires that (a) we wait till the system has reached equilibrium before starting averaging, and (b) that we average over many independent microstates after we reach equilbrium. Both these requirements mean that

the increasing the speed at which we can evolve the system is critical so that we can generate many thousands of time steps in a reasonable time.

Since this is Python, the majority of the speed gains we can achieve are by vectorisation. Vectorising the Metropolis-Hastings algorithm takes some thought, but can be done. One way to do this is to notice that, if we imagine mapping the cells of the lattice on to a chess board pattern, then all the black squares can be evolved at the same time, because evolving one black square does not affect the nearest-neighbours of any of the other black squares. The same is true of the white squares, so in principle we can do one "time step" by doing all the black squares, then all the white squares. Accessing all the black squares or all the white squares simultaneously can be done by generating an array of boolean values to "mask" certain lattice sites. Read up on numpy boolean masking to understand how this works!

The `numpy.roll()` function is also useful for vectorising as it allows us to get one of the nearest-neighbour spins for all cells in the lattice in a single function call.

More information on Metropolis-Hastings and other algorithms for solving the Ising model can be found at:

https://hef.ru.nl/~tbudd/mct/lectures/markov_chain_monte_carlo.html

https://hef.ru.nl/~tbudd/mct/lectures/cluster_algorithms.html

This exercise is based on an exercise developed by one of the demonstrators on this course, Danny van der Haven.