Phase transitions

9.1 Learning outcomes

Mathematics and Physics

- Markov chain Monte-Carlo integration.
- Phase transitions.

Computing and Python

- Using NumPy for data storage.
- Creating live plots while simulating.

9.2 Ferromagnets

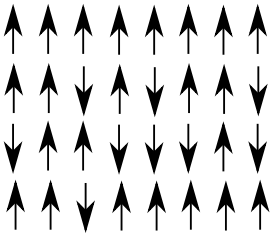
Ferromagnetism is the phenomenon that certain materials exhibit permanent magnetisation. At a microscopic level, the fact that a material behaves like a magnet has to do with the properties of the electrons 'orbiting' atom nuclei. Electrons have a small magnetic dipole moment, which means simply that every single electron behaves like a tiny magnet themselves. These little magnets wiggle around like crazy when the system is at nonzero temperature. However, if on *average* there are many more of these magnets pointing in one direction than in the other, a macroscopically large magnetic field can result.

You can create this average alignment in a piece of iron by putting it in a magnetic field. If the temperature is sufficiently low, and the electrons are thus not wiggling too much, the magnetic field will on average align them all in the same direction. You can then take the magnetic field away, and the alignment will remain, producing a permanent magnet. For the purpose of modelling this, we will thus have to worry only about electrons pointing either in the direction of the magnetic field, or opposite it. The individual electrons are tied to the atoms to which they belong, and in our model we will assume that the atoms are sitting at fixed positions on a rectangular lattice, like in the figure on the right.

A 1-dimensional model of a ferromagnet, with the tiny magnets formed by electrons pointing either up or down. In reality, the electrons flip their state frantically all the time, and this static picture gives a bad impression of what really happens. Each spins has 2 neighbours.

So this is all that we need for our model: a lattice of tiny magnets pointing either up or down, fluctuating under the influence of temperature. When on average they point more down than up, a macroscopic magnet results. We will consider 2 models: a 1 dimensional one, which is a toy model, and a 2-dimensional one. We will then see that they exhibits very different properties.

Now you may know that when you heat up a magnet to very high temperatures, at some point the magnetism gets lost. What we intuitively expect to happen is that due to the high temperature, the electrons wiggle around more and more violently, and at some point there is so much wiggling that on average they point as much up as down. The interesting thing is that this change in the magnetisation is rather sudden. At a specific temperature, the magnetisation can suddenly drops to zero very rapidly. This sudden change is called a *phase transition*. You are already familiar with other phase transitions: when water freezes the water molecules adopt a very different configuration: solid instead of liquid. This configuration changes occurs at 0° Celcius and is very sharp: at $T = 0.1^\circ$ the water is liquid and at $T = -0.1^\circ$ the water is solid. Spins systems are the simplest models to understand phase transitions hence why we have chosen it for this practical.



A 2-dimensional model of a ferromagnet, with the tiny magnets formed by electrons pointing either up or down. Each spins has 4 neighbours.

If you want to understand this from the simple model, you have to somehow look at the average configuration of all the magnets. Thermodynamics makes precise what this means.

It states that, if you fix the temperature, the probability P of finding, at any moment in time, a certain configuration of tiny electron magnets is

$$P(\text{configuration}) \propto \exp\left[-\frac{1}{k_B T} E(\text{configuration})\right] . \tag{9.1}$$

Here E is the energy of the configuration, k_B is the Boltzmann constant, called after the physicist who discovered this formula, and T is the temperature. So if you want to compute the average of the magnetisation M at a given temperature, as produced by all the tiny magnets together, you have to compute the total magnetisation for every single possible configuration, and sum over those configurations weighted by the probability above,

$$\langle M \rangle \propto \sum_{\text{configs}} \exp\left[-\frac{1}{k_B T} E(\text{config})\right] M(\text{config}) . \tag{9.2}$$

You could then study how this behaves as you change T. That sounds easy enough, ready to be put on a computer.

Except, the number of possible configurations of a system of only moderately few spins is enormous. We have two possible ways to align each tiny magnet, so for a 1dimensional lattice of 100 sites we end up with $2^{100} \approx 1.3 \cdot 10^{30}$ different possible configurations. For a 2-dimensional lattice of 100×100 sites, on the other hand, we end up with a whopping $2^{100\times 100} \approx 2 \cdot 10^{3010}$ different possible configurations, a number many orders of magnitude larger than the estimated number of particles in the universe! And here we are even lucky, because we have simplified our model so much that every tiny magnet can have only one of two different states. If you had a continuum of possibilities for every magnet, you would have a $2^{100\times 100}$ dimensional *integral*! There is *no* way we are going to be able to do these sums or integrals in a computer program by brute force. So before we can attack this problem, we have to learn more clever ways to compute averages over extremely large numbers of configurations.

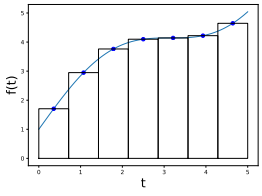
9.3 Monte-Carlo integration with the Metropolis algorithm

The way out, as usual, is to turn to gambling. Before we do that with large systems, let us see how the logic works by considering a simple one-dimensional integral. Let us assume that x describes the state of a particular system, and $\hat{P}(x)$ the probability that this state occurs. If we want to compute the average of some function of $f(x)$ over all possible configurations, this means we would have to compute

$$I = \int_{-\infty}^{\infty} \hat{P}(x) f(x)\, \mathrm{d}x . \tag{9.3}$$

For example, let us assume that the probability is given by

$$\hat{P}(x) = \frac{e^{-x^2}}{\displaystyle\int_{-\infty}^{\infty} e^{-y^2}\, \mathrm{d}y} . \tag{9.4}$$



Numerical integration: approximation of the area under the curve using rectangles.

and that we want to determine the average of a function $f(x) = x^2$ given the probability distribution above. The standard way to do a numerical integral of this type is something like the following:

file: `simple.py`

```
1 import numpy as np
2 import math
3
4 def P(x):
5     return math.exp(-x**2)/math.sqrt(math.pi)
6
7 def f(x):
```

```
    return x**2

N  = 100
xL = -10.  # 10 is close enough to infinity
xR =  10.

xvals = np.linspace(xL, xR, N)
fsum  = 0
for x in xvals:
    fsum += f(x)*P(x)

print(fsum/N*(xR-xL))
```

Problem 9.1:

Run this code and explain why it computes the integral (9.3). Compare the answer to the analytic one

$$\int_{-\infty}^{\infty} x^2 \frac{e^{-x^2}}{\sqrt{\pi}} = \frac{1}{2}\,. \tag{9.5}$$

(Bonus points if you can derive this result as well).

There are two problems with this approach when we want to generalise this to systems like the ferromagnet. One is that for a system with N degrees of freedom you end up, as emphasised earlier, with an N-dimensional integral, and even if you use only few steps per integral, the total number of function evaluations quickly becomes enormous. The other problem is that, typically, the probability density is not as simple as in (9.4), and it is impossible to determine the proper normalisation (the denominator in (9.4)).

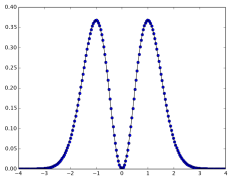
Let us start with the second problem. If you do not know the normalisation of the probability density, then you can of course try to compute it numerically as well, by evaluating the denominator of (9.4) numerically.

Problem 9.2:

Copy the program `simple.py` into `mc.py` and remove the division by `math.sqrt(math.pi)`, in the definition of `P(x)` so that the probability density is no longer properly normalised; we will call it $P(x)$ instead of $\hat{P}(x)$ from now on. Use the same logic to compute $\int_{-\infty}^{\infty} P(x)f(x)\mathrm{d}x$ as well as $\int_{-\infty}^{\infty} P(x)\mathrm{d}x$, and from that the original integral:

$$I = \frac{\int_{-\infty}^{\infty} P(x)f(x)\,\mathrm{d}x}{\int_{-\infty}^{\infty} P(x)\,\mathrm{d}x}\,. \tag{9.6}$$

Make a plot of the integrand $P(x)f(x)$ as a function of x, and superimpose on this a dot-plot which shows where you evaluate this integrand, something like in the figure on the right.



Function evaluations for ordinary numerical integration with regular sampling ($N = 100$).

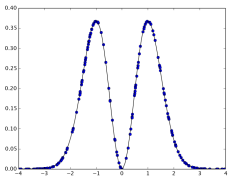
Check Point 9.1

What you have been doing so far is known as "regular sampling": split up the integration region into equally-spaced segments, and evaluate $P(x)f(x)$ for each of those regions. That is, however, not really necessary. Instead of evaluating the integrand at regularly spaced intervals, you can also just evaluate it at random points. This is called *Monte-Carlo integration*.

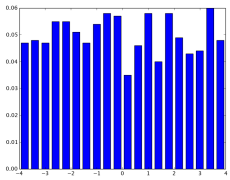
Problem 9.3:

Add code to your program which, instead of evaluating the integrand at regularly spaced intervals, simply picks `N` random points on the x-axis. Make a plot similar to the one of the previous task; this should look something like the figure on the right.

To generate random numbers you must import the module `random`. Then you can use the function `random.random()` which returns a uniform random number in the interval



Function evaluations for Monte-Carlo integration with unbiased sampling ($N = 100$).



Distribution of x-values for $N = 1000$: more or less flat.

$(0, 1]$ or `random.uniform(a,b)` which returns a uniform random number in the interval $(a, b]$. We suggest you use random points in the interval $(-4, 4)$.

Add further code to your plot which collects all the values where you evaluate the integrand in a list `xvals`. You can then show the distribution of these values in a histogram using

```
hist, bins = np.histogram(xvals, 20)
hist = hist*1.0/N
width = 0.7 * (bins[1] - bins[0])
center = (bins[:-1] + bins[1:]) / 2
plt.bar(center, hist, align='center', width=width)
plt.show()
```

As your plot for the distribution of points shows once more explicitly, we are sampling the integrand $P(x)f(x)$ in an unbiased way: every point on the x-axis has the same probability of being chosen.

Check Point 9.2

This suggests yet another way to do the integral: instead of sampling points x with a flat probability, why don't we sample them from a distribution which follows $\hat{P}(x)$? Then, instead of evaluating

$$I = \frac{\frac{1}{N}\sum_{i=1}^{N} P(x_i)f(x_i)}{\frac{1}{N}\sum_{i=1}^{N} P(x_i)}, \quad \text{with } x_i \text{ chosen with equal probability,} \tag{9.7}$$

we can approximate the integral by simply evaluating

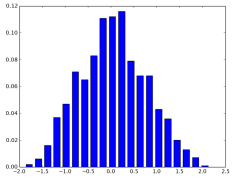
$$I = \frac{1}{N}\sum_{i=1}^{N} f(x_i), \quad \text{with } x_i \text{ distributed according to } \hat{P}(x_i). \tag{9.8}$$

This is a very useful thing to do when the probability distribution is strongly peaked for only a small range of values x_i. In that case, the unbiased sampling method wastes a lot of function evaluations to compute the integrand at points which do not contribute substantially to the integral. The sampling method (9.8) instead spends most of its time where it should spend it: where $\hat{P}(x_i)$ is large. For our ferromagnet problem, this will be very important: instead of sampling a huge configuration space, we will only look at configurations for which the probability is substantial.

The main problem with this is now, of course, to devise a method which produces points x_i such that their distribution follows $\hat{P}(x_i)$. The algorithm which does this is known as the *Metropolis algorithm*, and you will implement it below. The idea is that we start with some random point x, and then generate new ones by the following algorithm:

Metropolis algorithm

- Make a random step Δx chosen with `random.uniform(-0.5,0.5)`, to get x_{new}.
- If $P(x_{\text{new}}) > P(x)$, accept this new point.
- If $P(x_{\text{new}}) < P(x)$, accept this new point with probability $P(x_{\text{new}})/P(x)$. **If the new point is not accepted, use the old point again**.



Distribution of points according to (9.4) (for $N = 1000$).

It can be shown that this produces points x_i which are distributed exactly according to $\hat{P}(x_i)$.

Problem 9.4:

Implement the algorithm above, based on the following code template (you will need to combine it with what you have already written).

```
xprev = -0.2
xlist=[[xprev, f(xprev)]]
fsum  = f(xprev)
Pprev  = P(xprev)
dx = 0.5
for i in range(N):
    xnew = xprev + random.uniform(-dx, dx)
    Pnew = P(xnew)

    # Reject if Pnew < Pprev and Metropolis criterion does not hold.
    [FILL THIS IN]

    xlist.append([xnew,P(xnew)*f(xnew)])
    xprev=xnew
    Pprev=Pnew
    fsum += f(xnew)

xvals, pfvals = zip(*xlist)
```

You will notice that the answer `fsum/N` is not as close to 0.5 as with earlier methods; more on this later.

Again make a histogram plot of the distribution of the x_i values. Observe that the distribution now follows $\exp(-x^2)$, as in the figure.

Check Point 9.3

While the result for this type of integration is not as accurate in the one-dimensional case as the other methods, it has the advantage that it scales much better to problems where the number of configurations ('the number of points x_i') is extremely large. Because the Metropolis Monte-Carlo algorithm focusses attention to regions where the probability is substantial, large numbers of configurations are ignored automatically, thus cutting down the integration/summation problem substantially.

Problem 9.5:

Write a new program, called `mc2.py`, which integrates, using Metropolis Monte-Carlo, the function $f(x,y) = x^2 + y^2$ against the probability distribution $P(x,y) = \exp\left[-x^2 - y^2\right]$. This is the two-dimensional analogue of the problem studied earlier.

This is a 2-dimensional integral so you will have to generate random values of x and y independently, but the algorithm remains exactly the same. When you define `P(x)` and `f(x)` assume that `x` is a numpy array with 2 components. You must also be careful that when you write `a = b` in python, with `b` being a list, or an array, python does not copy the list or the array but makes a reference to it. This means that modifying `a` also modifies `b`! To ensure you make a copy, you can use `a = list(b)` for list and `a = np.array(b)` for arrays. For a general solution, load the module `copy` and use `a = copy.copy(b)`.

This time, instead of drawing a histogram of the random point, make a plot of the points (x,y) (draw a `"o"` and use the plot argument `markersize=0.1`) at which you ended up evaluating the integrand, and observe that the density of points follows $\tilde{P}(x,y)$.

Note: You can compute the result analytically with Python as well, using

file: `analytic.py`

```
from sympy import *

x,y = var('x, y')
num = integrate( (x**2+y**2)*exp(-x**2-y**2), (x,-oo,oo), (y,-oo,oo))
den = integrate( exp(-x**2-y**2), (x,-oo,oo), (y,-oo,oo))

print("numerator   = " + str(num))
print("denominator = " + str(den))
print("integral    = " + str(num/den))
```

The `sympy` module provides all sorts of functionality to do computations (integrals, derivatives, series and so on) *analytically* instead of numerically. See `http://sympy.org` for more information.

Check Point 9.4

With the problem of integrating (or summing) over large numbers of configurations out of the way, we can now turn to our original problem of interest, the ferromagnet.

9.4 Simulating lattice spins

The name comes from Ernst Ising, a German student who was given the task to solve the one-dimensional model analytically by his supervisor Wilhelm Lenz, in 1924. We will look at the two-dimensional version, which is much harder to deal with analytically.

Let us briefly recall our model of the ferromagnet. This model is the so-called *Ising* model. It postulates that at a microscopic scale, magnetism is caused by spinning electrons, which are effectively little magnets which point either 'up' or 'down' (or more accurately: in the direction of an externally applied magnetic field or in the opposite direction). These "spins", as they are often called in the literature, are fixed on a regular two-dimensional lattice. If we fix the temperature T to some value, then the probability of finding a particular configuration of spins is determined by the energy of this configuration, through (9.1).

What we should therefore now do first is to look more closely at the energy of this system. In 1 dimension, we will denote the spin of an electron at site i by σ_i; this variable can take the value $+1$ or -1 for 'up' and 'down' respectively. We will now assume that the energy of the system only involves so-called *nearest neighbour* interactions. That is, when we write down the energy formula, it only contains terms which involve two spins which are directly to the left/right or top/bottom of each other. The formula reads

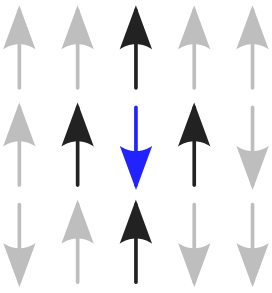
$$\begin{aligned} E &= -J\frac{1}{2}\sum_i \sigma_i\left(\sigma_{i+1}+\sigma_{i-1}\right) - h\sum_i \sigma_i \\ &= -J\sum_i \sigma_i\sigma_{i+1} - h\sum_i \sigma_i \end{aligned} \tag{9.9}$$

The 1-dimensional nearest neighbours (dark arrows) of the central spin (blue).

Here i run over the labels of all spins on the lattice. We also use periodic boundary conditions meaning that if the lattice has N sites i each run from 0 to $N-1$ and then $\sigma_N = \sigma_0$. The parameter J measures the strength of the interaction between two neighbouring spins, and h denotes the strength of an externally applied magnetic field.

In 2 dimensions we will denote the spin of an electron at site i,j by $\sigma_{i,j}$ and the energy will then become

$$\begin{aligned} E &= -J\frac{1}{2}\sum_{i,j} \sigma_{i,j}\left(\sigma_{i+1,j}+\sigma_{i-1,j}+\sigma_{i,j+1}+\sigma_{i,j-1}\right) - h\sum_{i,j} \sigma_{i,j} \\ &= -J\sum_{i,j} \sigma_{i,j}\left(\sigma_{i+1,j}+\sigma_{i,j+1}\right) - h\sum_{i,j} \sigma_{i,j} \end{aligned} \tag{9.10}$$



The 1-dimensional nearest neighbours (dark arrows) of the central spin (blue).

Here i,j run over the labels of all spins on the lattice. In 2-dimensions periodic boundary conditions means that if the lattice is $N\times N$ then i,j each run from 0 to $N-1$ and then $\sigma_{N,j}=\sigma_{0,j}$, $\sigma_{-1,j}=\sigma_{N-1,j}$, $\sigma_{i,N}=\sigma_{i,0}$ and $\sigma_{i,-1}=\sigma_{i,N-1}$.

Problem 9.6:

The `Spins_1d` class defined in `spins_1d.py` contains some setup code and some plotting helpers for later.

file: `spins_1d.py`

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import random
4
5 class Spins_1d(object):
6     """ A class to simulatte spin dynamics on a 1 dimentional lattice
```

```
    """

    def __init__(self, N, J=1.0, h=0.0, T=1):
        """
        : param N : lattice size is N
        : param J : interaction strength
        : param h : external magnetic field
        : param T : temperature
        """
        self.N = N
        self.J = J
        self.h = h
        self.T = T
        self.setup_lattice()
        self.setup_plotting()

    def setup_lattice(self):
        """ Setup the lattice with a configuration of all-up spins.
        """
        self.lattice = np.ones([self.N])

    def setup_plotting(self):
        """ Setup a figure for interactive plotting.
        """
        plt.close("all")
        self.live = plt.figure(figsize=(4, 1))
        plt.ion()
        plt.axis('off')
        plt.show()

    def plot_lattice(self, thermalising=False):
        """ Plot the spin configuration.
        : param thermalising : if true display lattice in grey instead
            of blue
        """
        X, Y = np.meshgrid(range(self.N+1), range(2))
        cm = plt.cm.Blues
        if thermalising:
            cm = plt.cm.Greys
        # convert 1 dim array to 2 dim array
        lat = np.zeros([1,self.N])
        lat[0,:] = self.lattice
        plt.pcolormesh(X, Y, lat, cmap=cm)
        plt.axis('off')
        plt.draw()
        plt.pause(0.01)
        #plt.cla()

    def plot_magnetisation(self, T_E_M_values):
        """
        Plot the magnetisation as a function of the temperature.
        : param T_E_M_values : list of tuples (T, E, M)
        """
        plt.close("all")
        #plt.rc('text', usetex=True)
        plt.rc('font', family='serif')
        plt.ioff()
        plt.axis('on')
        plt.xlabel('$T$')
        plt.ylabel('$M$')
        plt.title('YOUR TITLE HERE')
        # plot M as a function of T from the data in T_E_M_values

if __name__=="__main__":
      # add all your testing code here
      s = Spins_1d(5)
```

Add to the class `Spins_1d` a member function `measure(self)` which computes the energy (9.9) and the total magnetisation of the current configuration. The total magnetisation is defined simply as the sum of the spin values over the entire lattice. When

`measure(self)` can be called as

```
s = Spins(20)
print(s.measure())
```

to print the energy and the total magnetisation of the current configuration. You will need to figure out a way to access elements of the lattice in such a way that periodic boundary conditions are implemented, i.e. that the lattice site N is the same as lattice site 0. Hint: use the 'modulo' operator '`%`'. (We can also use the numpy `roll` function here to make it faster, but as the `measure` function is not called often it does not matter much here.)

Looking at the energy (9.9), we notice that for a fixed value of i the only non-zero terms in the first sum are $\sigma_i\sigma_{i+1}$ and $\sigma_i\sigma_{i-1}$. Notice that to avoid counting each term twice, you only need to compute the terms $\sigma_i\sigma_{i+1}$ as the other will be computed when you consider the points $i-1$.

Test your code by looking carefully at how the lattice gets initialised (in `setup_lattice`), and then determine what the value of E and M should be for this $N = 20$ system (with $J = 1$, $h = 0$, as is the default). Do not continue unless you get the correct answers.

With the basic structure set up, we now want to determine the expected energy and magnetisation given a fixed temperature. For this we will have to average over all possible configurations of spins, weighted with the probability (9.1) with energy (9.9). This is clearly a large multiple sum (there are 2^N possible configurations for a 1-dimensional N lattice), so we certainly do not want to simply sum over all those configurations. Instead, we use a Monte-Carlo sum, in which we generate a subset of the possible configurations using a random process, such that their probability distribution approaches (9.1) with (9.9).

We do this, just as in task 9.4, by starting from an arbitrary configuration (typically: all spins up) and then generating a new one by randomly flipping one spin. This new configuration will then be accepted when *either* the probability for this new configuration is higher than for the old configuration, *or* when the Metropolis test holds. The criterion for *acceptance* will thus be that

$$(E_{\text{new}} - E_{\text{old}}) < 0 \quad \text{or} \quad \text{random.random()} < \exp\left[-\frac{1}{k_B T}(E_{\text{new}} - E_{\text{old}})\right] . \tag{9.11}$$

Clearly, we need to be able to compute the energy differences $E_{\text{new}} - E_{\text{old}}$ first. In our program the variable `T` will stand for the energy $k_B T$.

Problem 9.7:

How does the magnetisation $\sum_i \sigma_i$ of a configuration change if you flip a single spin σ_p? How does the energy (9.9), $\delta E = E_{\text{new}} - E_{\text{old}}$, change for that flip? Check your answer with us before you proceed to the next problem.

Check Point 9.5

Problem 9.8:

Write a member function `change(self, pos)` which computes the change of the energy and the change of the magnetisation when a single spin at position `pos` would be flipped. The function must return $\delta E = E_{\text{new}} - E_{\text{old}}$ and $\delta M = M_{\text{new}} - M_{\text{old}}$ as a tuple. The function must not call the class function `measure` as this would make it far too slow.

Test your code by evaluating the quantities ΔE, ΔM on a lattice for which you can easily compute the expected result, e.g. a lattice with all spins up:

```
s = Spins(5)
E,M   = s.measure()
dE,dM = s.change( 3 )
print E, M, dE, dM
```

What should the answer be? (again, use $J = 1$, $h = 0$)

Problem 9.9:

Now write a function `monte_carlo(self, steps)` which runs the Monte-Carlo algorithm for `steps` iterations, and returns the average energy and average *absolute* value of the magnetisation.

The function must first compute the energy and magnetisation of the initial configuration and store their values in `E` and `M`.

The algorithm is the same as before. You must start by selecting a random point on the lattice, for example by using the function `np.random.randint(0, N)` which returns a random integers, in the range $[0, N-1]$.

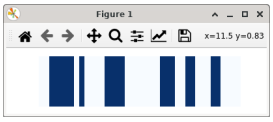
You must then compute by how much the energy of the lattice changes if you flip the configuration of that lattice site (you have just written a function which does it). You must then use (9.11) to decide if you flip the spin at the lattice site or not, and when you do, do flip the spin. Notice that the temperature is given by the class variable `self.T` (and we assume $k_b = 1$ for now).

When you flip the spin, you must also modify `E` and `M` by the amount that they have changed so that they always correspond to the energy and magnetisation of the current configuration. For debugging, set the temperature `T` to the value 2.5.

If you add a call

```
if n%2000==0:
    self.plot_lattice(false)
```

somewhere inside your main loop, you will get a live display (every 2000 updates) of the Monte-Carlo process as it runs, which may help with debugging.[1]



Live visualisation of the Monte-Carlo updates.

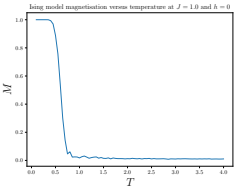
Check Point 9.6

Problem 9.10:

Write a member function `simulate(self, temperatures, steps)`, which takes a list of values for k_BT in 'temperatures', and runs the Monte-Carlo process for 'steps' steps. It should return a list of tuples (T, E, M). For simplicity we take $k_B = 1$. This is equivalent to consider that `T` stands for the energy k_BT.

Complete the member function `plot_magnetisation(self, T_E_M_values)` which takes the output of `simulate` and produces a plot of the magnetisation against the temperature. The `zip` function is useful to convert the list of tuple values `(T,E,M)` into 3 separate lists containing the values of `T`, `E` and `M`.

Add code to your `if __name__=="__main__":` block which calls this member function for a scan over 40 temperatures between $k_BT = 0.1 \ldots 4.0$, running at every temperature for $200\,000$ steps, and prints the average energy and average absolute magnetisation in tabular form (this takes a while; debug your code with fewer steps). Then call the member function `plot_magnetisation` to display the result. You should find something like the figure on the right.



The magnetisation as a function of the temperature for the two-dimensional Ising model. This was obtained on a lattice with $N = 10000$ and `steps=1000000`.

You should run the program first with $N = 100$ to check that it works. Then take $N = 40000$ and set the class variable `live-show` to `false`.

You can now run the suplied program `spin_1d.py` to generate the corresponding temperature dependance of the magnetisation and you should get a figure similar to

Check Point 9.7

The critical temperature for the two-dimensional model, where the magnetisation suddenly drops to zero, was in fact solved analytically in 1944 by Lars Onsager [1] (without computer simulations of course), and found to be

$$k_BT_c = \frac{2}{\ln\left(1+\sqrt{2}\right)} \approx 2.27\,. \tag{9.12}$$

[1] If you use Spyder, you need to set it up such that it uses a separate window for plots. Go to "Tools > Preferences > IPython console > Graphics > Backend" and change "Inline" to "Qt5" or "Gtk3". Then restart Spyder.

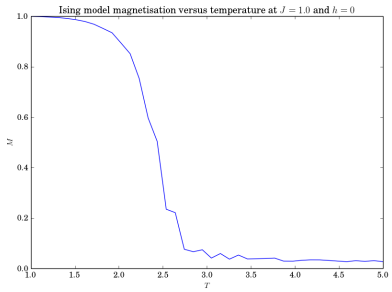


Figure 9.1: Magnetisation as a function of Temperature for the 2 dimensional model.

On the other hand, it was already known that there are no phase transition for the 1 dimensional model. This should agree (roughly, there are quite a few sources of error) with your simulations.

Problem 9.11:

A different way to initialise the lattice of spins is to use the following line in the method `setup_lattice()`:

```
self.lattice = np.random.choice([-1,1], [self.N])
```

Run your code with this initial configuration for a few low temperatures and observe the difference with respect to the runs you did earlier. Do you understand what you see?

When you were doing task 9.4, you may have played with the starting point for the Monte-Carlo process (there set to $x = -0.2$). You may have noticed that when you put the starting point far away from where the probability density P is highest, the algorithm needs a while before it has walked to the peak of P. While it does that, it mis-estimates the integral, and you typically get a far too large result.

For our Ising model, something similar happens: the configuration with all spins up (which we start out from) may not be one that has a particularly large probability. Therefore, it may take a while before the algorithm has flipped a sufficient number of spins before it reaches a configuration which has a high probability. In order to 'cut out' that part, we can introduce the concept of a *thermalisation step*: step for e.g. $(1 + f) \times$ steps, with f some number, but only start computing the average energy and magnetisation after the first $f \times$ steps have been taken.

Problem 9.12:

Add a parameter `self.f` to your `Spins` class and adjust the `monte_carlo` method so that it does $(1 + f) \times$ steps steps in total, but only starts computing the E and M average once the first f steps have been made. If you replace the bit that calls the visualisation routine with

```
if n%2000==0:
    self.plot_lattice( n < self.f * steps)
```

the thermalisation steps will be visualised in grey rather than blue.

The following program should then produce two plots, one without and the other with thermalisation.

file: `test_thermal.py`

```

import spins_1d as spins
import numpy as np

sp = spins.Spins_1d(1000, J=1.0, h=0.0, f=0.2)
```



```
results = sp.simulate( np.linspace(1, 3.0, 41), 200000)
sp.plot_magnetisation(results)

sp = spins.Spins_1d(1000, J=1.0, h=0.0, f=0)
results = sp.simulate( np.linspace(1, 3.0, 41), 200000)
sp.plot_magnetisation(results)
```

You should notice that the plot *with* thermalisation steps is smoother than the one without.

Check Point 9.8

9.5 Extra problems

Problem 9.13:

Use the Monte-Carlo integration technique to write a program which approximates π numerically by determining the area of a disc of radius 1. Submit `pi.py`; its output should be the numerical estimate of π. Use a function which is 1 inside the circle of radius $R = 1$ and 0 outside.

9.6 References

[1] Lars Onsager. "Crystal statistics. I. A two-dimensional model with an order-disorder transition". In: *Phys.Rev.* (1944). http://link.aps.org/doi/10.1103/PhysRev.65.117, pp. 117–149.

