PHYS3020 - Computational Assignment

Tom Stephen, 45811449

October 27, 2022

1 1D Ising Model

1.1 Demo

Noting that there is a lot of overlap in what is required for the 1D case in this question, then the 2D and 3D cases later on, I implemented some classes for a general simulation (which handles some shared initialization, time averages, and various quantities), which is then inherited by sub classes, specific for each of the different dimension-counts. The code for this is included in section 4.1 For each, there are some subtle difference in computing total and local energy, stepping the simulation, etc, which I will discuss later.

To (hopefully) demonstrate that the one-dimension implementation of the Ising model is working, we show an example equilibria state for three different temperatures after 200,000 steps in figure 1. The code to produce this is included in section 4.3. This involves initialising each of the states with a random distribution of spins, then applying the metropolis algorithm - choosing a dipole at random, and flip it if either it decreases the energy of the system, or by some random chance given by the exponential of the temperature. This random chance is proportional to temperature, and so for higher temperatures the system is likely to be in a state of disarray, as dipoles will often be flipped irrespective of the energy change.

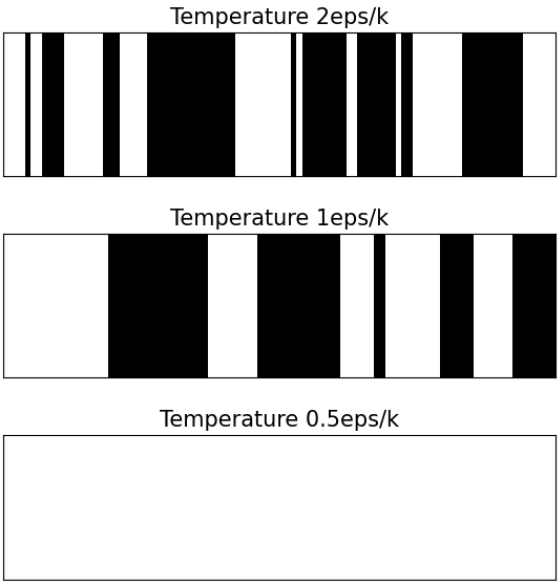

Figure 1: Equilibria results for the 1D Ising model, with $N = 100$ dipoles and after 200,000 spin-flip opportunities.

The number 200,000 was chosen as we wish to ensure each dipole has at minimum 1000 chances to flip, so

$$200,000 = \underbrace{100}_{\text{no. dipoles}} \times \underbrace{1000}_{\text{min. chances}} \times \underbrace{2}_{\text{safety}}$$

At lower temperatures, there are often only small, isolated "chunks" of anti-parallel dipoles, otherwise mostly consisting of the same spin. This is as compared to the higher temperatures, where there is much less order, and no clear dominance of one spin over another. This is a promising result - lower temperatures settle into order, and larger temperatures have enough energy to stay in a chaotic state, as expected.

1.2 Derivation

Given the partition function

$$Z = (2\cosh(\beta\epsilon))^N$$

we may derive the internal energy, free energy, entropy and specific heat of the system. First, the internal energy,

$$\begin{aligned}
U &= \frac{(kT)^2}{Z}\frac{\partial Z}{\partial kT} \\
&= \frac{(kT)^2}{Z}\left(-\frac{\epsilon 2^N N \sinh(\beta\epsilon)\cosh^{N-1}(\beta\epsilon)}{(kT)^2}\right) \\
&= -\frac{\epsilon 2^N N \sinh(\beta\epsilon)\cosh^{N-1}(\beta\epsilon)}{(2\cosh(\beta\epsilon))^N} \\
&= -\frac{\epsilon 2^N N \sinh(\beta\epsilon)\cosh^{N-1}(\beta\epsilon)}{2^N\cosh^N(\beta\epsilon)} \\
&= -\frac{\epsilon N \sinh(\beta\epsilon)\cosh^{N-1}(\beta\epsilon)}{\cosh^N(\beta\epsilon)} \\
&= -\frac{\epsilon N \sinh(\beta\epsilon)\cosh^{N}(\beta\epsilon)}{\cosh^N(\beta\epsilon)\cosh(\beta\epsilon)} \\
&= -\frac{\epsilon N \sinh(\beta\epsilon)}{\cosh(\beta\epsilon)} \\
&= -\epsilon N \tanh(\beta\epsilon)
\end{aligned}$$

then internal energy per dipole is

$$u = \frac{U}{N} = -\epsilon\tanh(\beta\epsilon)$$

as required.

Next, the free energy is given by

$$\begin{aligned}
F &= -\tau\ln Z \\
&= -\tau\ln\left((2\cosh(\beta\epsilon))^N\right) \\
&= -\tau N\ln\left(2\cosh(\beta\epsilon)\right) \\
&= -\tau N\ln\left(e^{\beta\epsilon}+e^{-\beta\epsilon}\right) \\
&= -\tau N\ln\left(e^{\beta\epsilon}\left(1+e^{-2\beta\epsilon}\right)\right) \\
&= -\tau N\ln e^{\beta\epsilon} - \tau N\ln\left(1+e^{-2\beta\epsilon}\right) \\
&= -\tau N\beta\epsilon - \tau N\ln\left(1+e^{-2\beta\epsilon}\right) \\
&= -N\epsilon - \tau N\ln\left(1+e^{-2\beta\epsilon}\right)
\end{aligned}$$

then, free energy per dipole,

$$= -\epsilon - kTN\ln\left(1+e^{-2\beta\epsilon}\right)$$

as required.

Next, the entropy is given by

$$\begin{aligned}
S &= \frac{U-F}{T} \\
&= \frac{-\epsilon N\tanh(\beta\epsilon)+\epsilon N + NkT\ln\left(1+e^{-2\beta\epsilon}\right)}{T} \\
&= \frac{\epsilon}{T}\left(N\tanh(\beta\epsilon)\right) + kN\ln\left(1+e^{-2\beta\epsilon}\right)
\end{aligned}$$

entropy per dipole,

$$= \frac{\epsilon}{T}\left(1-\tanh(\beta\epsilon)\right) + k\ln\left(1+e^{-2\beta\epsilon}\right)$$

as required.

Finally, there's a few different definitions we can use for the specific heat: we'll use $\frac{\partial U}{\partial T}$ since our result for U is fairly

short,

$$
\begin{aligned}
C_V &= \frac{\partial U}{\partial T} \\
&= \frac{\partial}{\partial T}\left(-\epsilon N \tanh(\beta\epsilon)\right) \\
&= -\epsilon N \frac{\partial}{\partial T}\left(\tanh\left(\frac{\epsilon}{kT}\right)\right) \\
&= -\frac{\epsilon^2 N}{kT^2}\operatorname{sech}\left(\frac{\epsilon}{kT}\right) \\
&= -\frac{\epsilon^2 N\beta}{T}\operatorname{sech}\left(\beta\epsilon\right) \\
&= -\frac{\epsilon^2 N\beta}{T\cosh^2(\beta\epsilon)}
\end{aligned}
$$

and again, specific heat capacity,

$$
\begin{aligned}
c &= \frac{C_V}{N} \\
&= -\frac{\epsilon^2\beta}{T\cosh^2(\beta\epsilon)}
\end{aligned}
$$

as required.

To use these theoretical relationships to compare against the simulations, I wrote the code included in section 4.2 which defines a function for each of the above quantities such that I can later iterate over a list of temperatures to fetch the value of the quantity at each point.

1.3 Verification

We now run some simulations over a range of temperatures to confirm the accuracy of the above theory by plotting it against results measured from the simulation. Specifically, this is done for internal energy per dipole, Helmholtz free energy per dipole, entropy per dipole, specific heat capacity and reduced magnetism per dipole. This is done using a Python script (see section 4.4), and produces the plots in figure 2.

The experimental results were tested at 20 different temperatures - each of these was a new simulation, run until we expected some equilibrium, and then a time average of the relevant quantities was taken. There is a general time average function defined in the Simulation super class,

```
def time_average(self, quantity):
    # excluded- if/elif/else statement to assign relevant class method to "func" variable

    vals = np.zeros(25)
    for i in range(len(vals)):
        self.step(steps = self.N)
        vals[i] = func()

    mean = np.mean(vals)
    err = np.std(vals)

    return mean, err
```

So taking the time average of some quantity is fairly quick[1] and easy in code. The details for each of the quantities follow directly from the statements in the theory of the assessment spec - detailed explanations are however included in the comments of the simulation classes.

[1] If horribly inefficient - really, each time I step the simulation forwards here, I should record all of the quantities, but the time complexity of stepping by N (which is only around 100) is relatively small compared to actually computing these quantities.

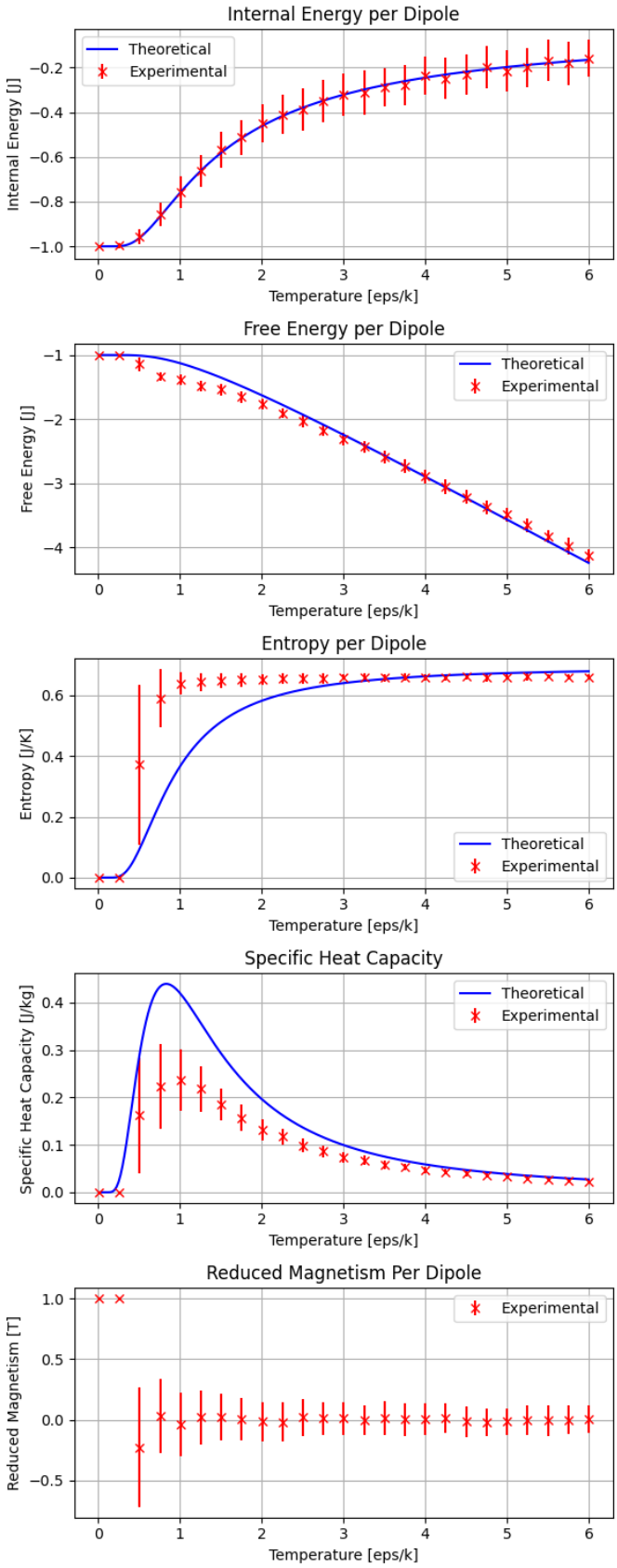

Figure 2: Comparison of theory-derived relationships for quantities as functions of temperature and experimentally measured results for a one-dimension Ising model simulation with 100 dipoles.

1.4 Analysis

For each of the quantities with both a theoretical and experimental result, the trends appear to be similar: experimental results for internal energy and free energy both closely agree with the theory. Although the peak in theoretical specific heat capacity is about twice that of the experimental results, the trend is very similar, with the peak occurring at the same temperature. The entropy per dipole shows the largest discrepancy, exhibiting a sudden change from almost no

entropy at low temperatures to a plateauing entropy at temperatures above $1\frac{\epsilon}{k}$, as compared to the more gradual change in theory.

In fact, this sharp change occurs around the same temperature as the peak in specific heat capacity, the reduction in reduced magnetism per dipole, and the beginning of the increase in internal energy per dipole. Thus, there is some 'phase change' or critical temperature here, such that the system displays vastly different characteristics either side. This is consistent with Fig. 1, which shows a relatively organised system at low temperatures (low multiplicity, and in turn entropy, often dipoles are parallel so energy is low, etc), then a disorganised system at high temperatures (so high multiplicity and in turn entropy, etc).

The magnetism plot in Fig. 2 is not precisely as expected - recall that the reduced magnetism is given by the mean dipole spin. For high temperatures, the system should be in such a state of disorder than there are roughly equal dipoles in spin-up vs spin-down, and so the mean will be zero, and this is observed. At low temperatures, lower than the observed critical temperature, the system should eventually settle into being either entirely spin-up or spin-down, again resulting in a mean result of zero (albeit with a large uncertainty, since all values will be either -1 or 1). Practically, however, it appears the system has a tendency[2] to settle into the spin-up state.

1.5 Histograms

To further investigate the reduced magnetism per dipole, we run two new sets of simulations: a simulation of a 100-dipole lattice is initialised at some temperature, and then run until we expect it has reached equilibrium. This is repeated 200 times, before presenting the reduced magnetism at the end of each simulation in a histogram. This is the repeated at 3 temperatures, and then again for the same temperatures but with a 500-dipole lattice. The script to complete this is included in Section 4.5, and the results presented in Fig. 3 and Fig. 4.

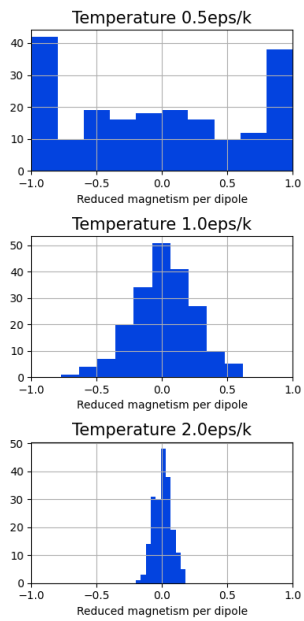

Figure 3: 100 dipoles

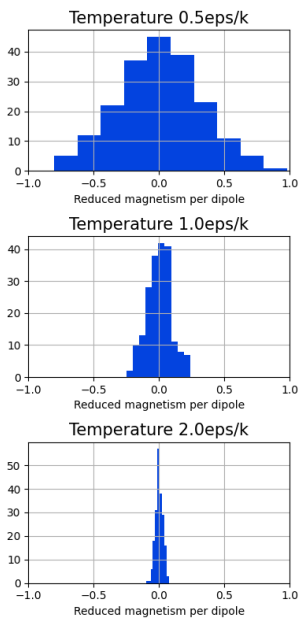

Figure 4: 500 dipoles

For the 100-dipole lattice, at low temperatures, we see that the system typically tends to a more magnetised state (that is, the mean absolute magnetism is greater), although the average magnetism is 0. As the temperature increases and the system becomes more disorganised, the mean magnetism tends towards zero. This is in contrast with the 500-dipole lattice, which displays a similar trend but with little magnetism at the lowest temperature.

Extending this, for an infinite lattice, we expect the temperature where magnetism polarises to decrease further and further until, at 0K, there will be no clear magnetised state. For any infinite lattice starting with a suitably random distribution of dipole spins, there will always be infinite up-spins and infinite down-spins, and hence no clear magnetism.

[2] or, at least, for the tests I ran - there is not so many repeat tests than this is functionally impossible with an even chance to settle into either final state

2 2D Ising Model

2.1 Extend Simulation

We now extend the simulation to cover two dimensions - doing so is fairly trivial by implementing a new sub class `TwoDSimulation()`, and redefining the functions to initialise a state (as we know require an $n \times n$ matrix of random values), compute the energy, compute the local energy difference, and stepping the Metropolis algorithm (however this is basically identical, besides using two randoms values each interation, one for either of the x and y coordinate.). The specific heat capacity also needs to be slightly modified, since we need to measure take the square energy rather than along a plane.

Again, the code for the simulation classes is included in Section 4.1. To visually check the model is working, we initialise 3 models and step them until equilibrium. This is done with the script in Section 4.6, and the output is included in Fig. 5.

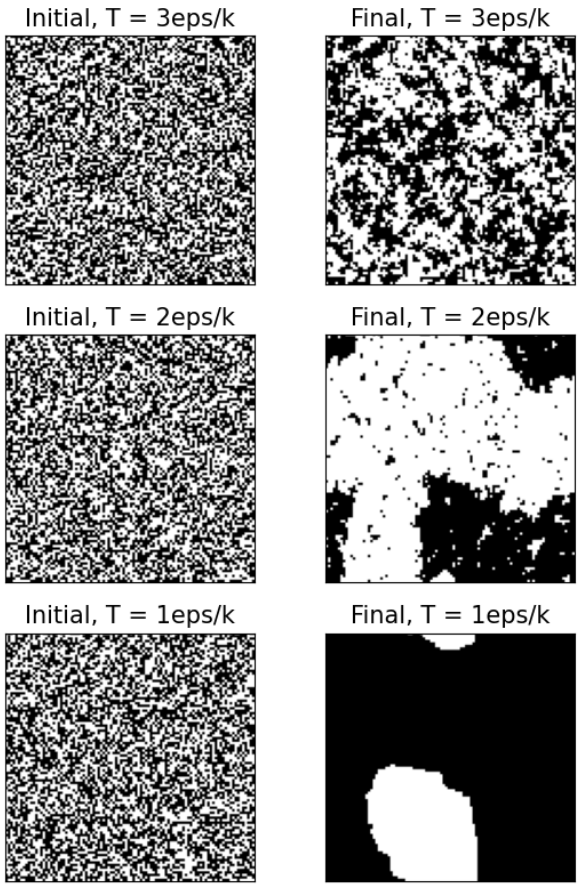

Figure 5: Initial and "equilibrium" states for 2D ising model of a 100×100 lattice, after 10,000,000 spin-flip opportunities. Repeated for 3 temperatures to show the effect temperature has on the likelihood of random spin-flips.

The initial states, shown on the left of Fig. 5, show that the system starts in a state of disorder, regardless of temperature. At $T = 3\epsilon/k$, small clusters of parallel dipoles begin to form, growing into a large chunk of similar dipoles at $T = 2\epsilon/k$, and finally decaying to only one large crystal at $T = 1\epsilon/k$. This shows the expected behavior of the model, both in turns of the physics (disorder proportional to temperature) and the code implementation (crystals and therefore neighboring dipole interactions must either be adjacent or cross the edge of the lattice, hence extending the lattice to be an approximation of infinite).

As a bonus, I've also created an animated gif of the above plot, see it here `https://media.giphy.com/media/KqAbSonkl8tmkgGYum/giphy.gif`.

2.2 Crystals

As seen in Fig. 5, the crystals appear to grow as the temperature settles - with the lower temperature, there is a smaller chance for energy-increasing spin flips, which mis-align neighboring dipoles.

To study whether there exists a critical temperature and perhaps propose what it may be, I have simulated the above but for 10 temperatures, over the relatively narrow range $1.7\epsilon/k \rightarrow 2.6\epsilon/k$. This was done with the script included in Section 4.7, and results presented in Fig. 6.

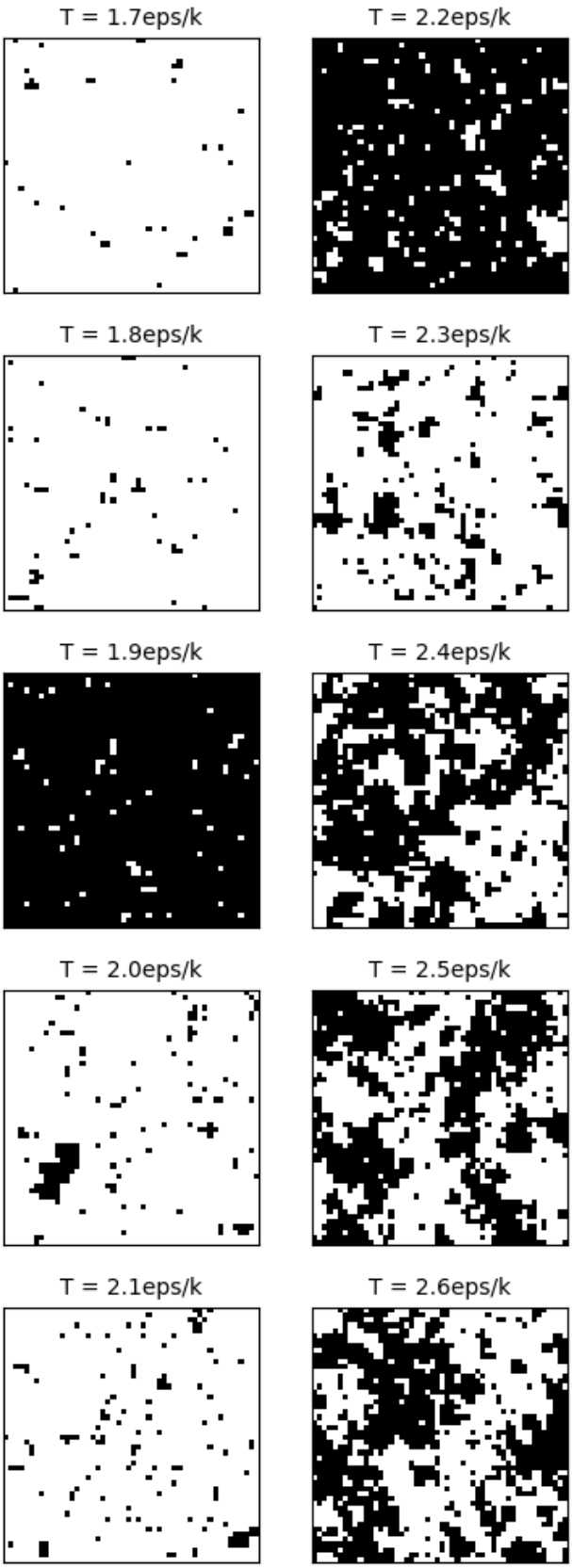

Figure 6: Critical temperature estimation by studying the existence of crystals in the final state of a simulation.

In Fig. 6, we see that for low temperatures the system settles to an almost homogenous state - there are a few outlying, but for the most part, it is entirely one state. As temperature increases, this is mostly consistent, until around $T = 2.4\epsilon/k$ where we suddenly observe some crystals forming, but an overall varied distribution of dipoles. This behavior is consistent as temperature continues to increase.

Hence, I estimate $T_c \approx 2.4\epsilon/k$ is the critical temperature for the 2d lattice.

2.3 Quantities

We know repeat the same process as section 1.3, but now with 2D lattices of dipoles. The partition function used to derive the theoretical quantities earlier is only applicable for the 1D case, so we present the experimental results for $N = 20^2$, $N = 50^2$ and $N = 100^2$ lattices instead. The computations are essentially identical, except the energy and specific heat capacity is summed along the x and y axis (rather than the single axis in the 1D case).

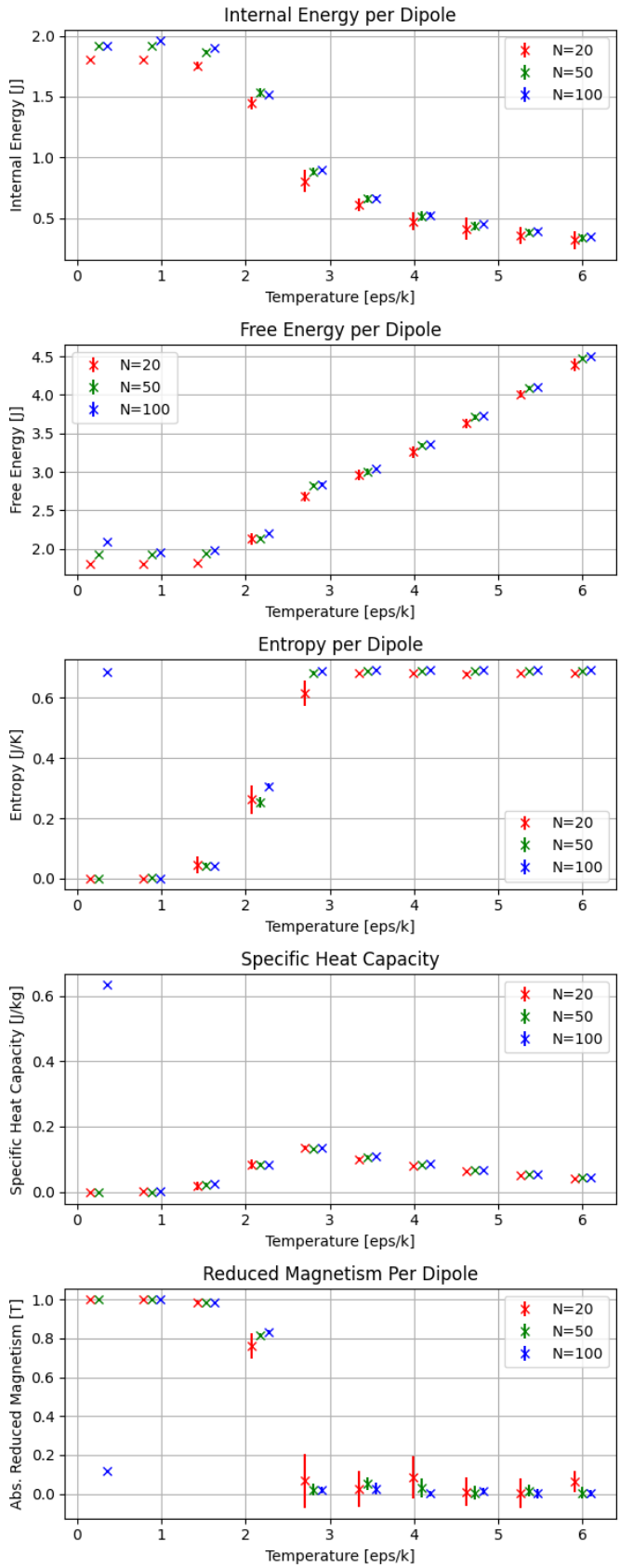

Figure 7: Comparison of quantities as function of temperature between lattices with differing dipole counts. The change in behavior around $T_C \approx 2.4\epsilon/k$ implies the existence of some critical temperature.

The results here appear similar to the results from figure 2, and individual trials appear similar between differing dipole counts. One key trend, particularly for the entropy and reduced magnetism, is that the uncertainty is much greater for the $N = 20^2$ cases as compared to the lattices with more dipoles. Over such a small lattice, having so few results means that any noise or variance will cause a much larger standard deviation and hence uncertainty.

Referring back to the previous prediction for a critical temperature, around $T_c = 2.4\epsilon/k$, we see that all quantities display some signficant change around this point - whether that be the internal energy beginning to decrease or free energy beginning to increase, or the sharp change in both entropy and reduced magnetism. The specific heat capacity also reaches a temporary maximum around this point. This observation reinforces the prediction that T_c exists and is approximately $2.4\epsilon/k$.

2.4 Mean positive/negative magnetism

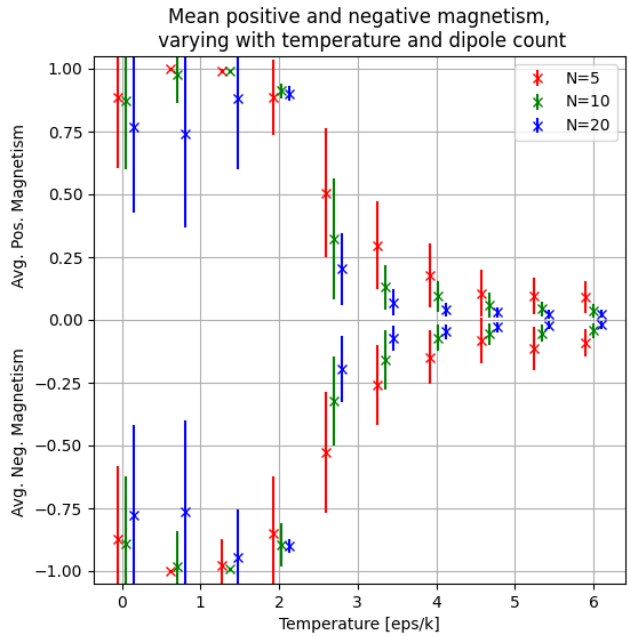

Figure 8: Mean positive and mean negative final magnetism of repeated runs, varying with temperature and dipole count. Again, change in behavior (specifically, diversification into two different magnetisms) suggests existence of critical temperature around $T_c = 2.4\epsilon/k$

We now set up many simulation runs, at both varying temperature and dipole count, and record the final, mean magnetism for each simulation. Then, for all of the simulations at the same temperature and dipole count, we record and present both the mean positive and mean negative magnetism. This is motivated by observing that as states cool down, they tend to settle into either spin-up or spin-down, arbitrarily. So, while the total mean magnetism is zero, we may show that the lattice settles at random into either of the two states.

This is presented in figure 8, and we see a clear match to the expected behavior. Error bars are large, but this is likely a combination of using a relatively small number of steps to reach "equilibrium" (as we need to run 100s of tests, a more definite equilibrium would take too long to reach. Perhaps, in the future, a method to look at the change in variance of the system over time could dynamically determine equilibrium state?), and the small dipoles. The error bars are not so large however to confidently state that at temperatures above the critical temperature, there is no clear magnetism, and below the critical temperature, the state will be magnetised into one of two polarities.

2.5 Heating and cooling

To allow for primitive heating a cooling of the 2D model, we add a method to the `TwoDSimulation` subclass,

```
class TwoDSimulation(Simulation):
    # excluded for brevity
    def set_temp(self, T):
        self.temperature = T * self.eps / self.k
        self.beta = 1 / (self.k * self.temperature)
```

So we may the adjust the temperature of an already defined/initialised model. The script to actually simulate and heat/cool the model is included in section 4.10, but the general process is

1. Initialise the simulation with temperature $T = 1\epsilon/k$

2. Step the sim until it reaches equilibrium
3. Record the current state
4. While the temperature is $< 3\epsilon/k$...
   (a) increment the temperature bby $0.1\epsilon/k$
   (b) update model temperature
   (c) step model minimum 10 times per dipole
5. Record the current state
6. While the temperature is $> 1\epsilon/k$
   (a) decrement the temperature by $0.05\epsilon/k$ (i.e., cool the model *slowly*)
   (b) update model temperature
   (c) step model minimum 10 times per dipole
7. Record the current state

Doing so, we arrive the figure 9. As expected, at equilibrium, the model is magnetised entirely one way. Heating the model introduces the disorder we'd expect for a hot model, erasing the cool state. Cooling the model slowly, one orientation takes precedence at random, and assuming that the model is cooled sufficiently slowly, we arrive at a complete magnetism. The difference between the first and final equilibria are arbitrary - these are simply the states the system happened to settle into.

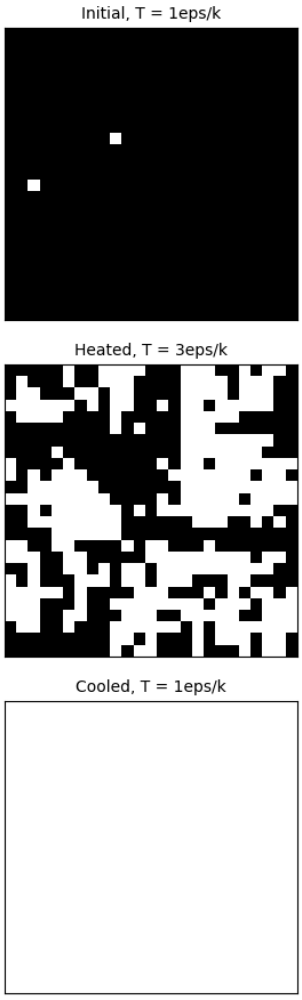

Figure 9: Demonstration of heating and slowly cooling simulation to change the magnetism of the state. Result is non-deterministic, so potentially must be repeated multiple times to arrive at the desired magnetism.

3 3D Ising Model

3.1 Implementation

The logical continuation and extension of this report, from the 1D and 2D Ising models, is to implement it once more in 3 dimensions. Due to the structure of the code, with the class/subclass system, this is a relatively simple procedure.

The complete class code for the 3D simulation is included with the other classes in section 4.1.

The initialisation, energy and local energy differnece all need to be extended to work in 3D dimensions. This is the same as the generalisation from 1D to 2D, simply doing the same calculations along yet another axis. Following the previous method for stepping the simulation however felt very slow, so `cProfile` was employed to determine what was taking so long. The profiling script is simple,

```
import cProfile
import simulation

sim = simulation.ThreeDSimulation(25)

cProfile.run('sim.step(steps = 25 * 25 * 10000)')
```

The output looks like the following,

```
~/Documents/UQ/PHYS3020/computational/code  python3 profiling.py
         11105364 function calls in 2.655 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    2.655    2.655 <string>:1(<module>)
  1875000    0.478    0.000    0.938    0.000 random.py:200(randrange)
  1875000    0.243    0.000    1.181    0.000 random.py:244(randint)
  1875000    0.336    0.000    0.460    0.000 random.py:250(_randbelow_with_getrandbits)
   625000    0.675    0.000    0.675    0.000 simulation.py:261(local_energy_difference)
        1    0.778    0.778    2.655    2.655 simulation.py:280(step)
        1    0.000    0.000    2.655    2.655 {built-in method builtins.exec}
  1875000    0.048    0.000    0.048    0.000 {method 'bit_length' of 'int' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
  2397831    0.076    0.000    0.076    0.000 {method 'getrandbits' of '_random.Random' objects}
   582529    0.021    0.000    0.021    0.000 {method 'random' of '_random.Random' objects}
```

So, for a piece of code that takes 2.655 seconds to run, 2.579s is spent generating random numbers choosing the random dipole! This is *horribly* inefficient, so we rewrite to instead generate matrix of random integers with dimensions 3 (for x, y, z) by the number of steps. We also transition from the default random library to use the numpy random library, which does the necessary work outside of Python, in some fast, compiled program (as far as I know, numpy's random library is implemented and optimised in C). Then, for each step, we simply index this random value matrix and proceed as before.

Below is a short script to show the before and after of this optimisation,

```
# === BEFORE ===
def step(self, steps):
    j = 0
    while j < steps:
        x = random.randint(0, self.width)
        y = random.randint(0, self.width)
        z = random.randint(0, self.width)

        # rest of step function...
# ==============

# === AFTER ===
def step(self, steps):
    j = 0
    rands = np.random.randint(self.width, size=(3, steps))
    while j < steps:
        x = rands[0, j]
        y = rands[1, j]
        z = rands[2, j]
```

```
        # rest of step function...
# ==============
```

3.2 Intuitive Testing

Before we attempt to measure some definite quantities, we design some visualization to check, intuitively, if it appears that the simulation is working correctly. This proved to be essential for me - I made a mistake in the bounds of choosing the dipole to test, so one corner of the 3D domain was fixed in the initial state, which propagated throughout the rest of the system. By viewing a 3D model of the system, this issue was easy to diagnose.

Hence, we begin similarly to how we started Q2a: we initialise three models, then allow them to run until equilibrium, and show the initial and final states to compare. The script to do this is below, and produces figure 10.

```python
import simulation
import matplotlib.pyplot as plt

# create grid of 3d projection subplots
# slightly frustrating to do this way - indexing subplots to
# remove and replace with the proper projection feels worse
fig = plt.figure()
axs = [ [0]*2 for i in range(3) ]
axs[0][0] = fig.add_subplot(3, 2, 1, projection='3d')
axs[0][1] = fig.add_subplot(3, 2, 2, projection='3d')
axs[1][0] = fig.add_subplot(3, 2, 3, projection='3d')
axs[1][1] = fig.add_subplot(3, 2, 4, projection='3d')
axs[2][0] = fig.add_subplot(3, 2, 5, projection='3d')
axs[2][1] = fig.add_subplot(3, 2, 6, projection='3d')
fig.set_size_inches(6, 8, forward=True)

# for temperature = 0.1, 1.5, 5 eps/k
for ind, temp in enumerate([0.1, 1.5, 5]):
    # init simulation
    sim = simulation.ThreeDSimulation(20, T = temp)

    # plot initial state
    axs[ind][0].voxels(sim.get_spins() + 1, facecolors=[0, 0, 1, 0.3])
    axs[ind][0].set_title("Initial, T = " + str(temp) + "eps/k", fontsize=8)

    # run sim until equilibrium
    sim.step(steps = 20 * 20 * 2000)

    # plot final state
    axs[ind][1].voxels(sim.get_spins() + 1, facecolors=[0, 0, 1, 0.3])
    axs[ind][1].set_title("Final, T = " + str(temp) + "eps/k", fontsize=8)

# tidy layout and save
fig.tight_layout()
plt.savefig("assets/q3viz.png")
```

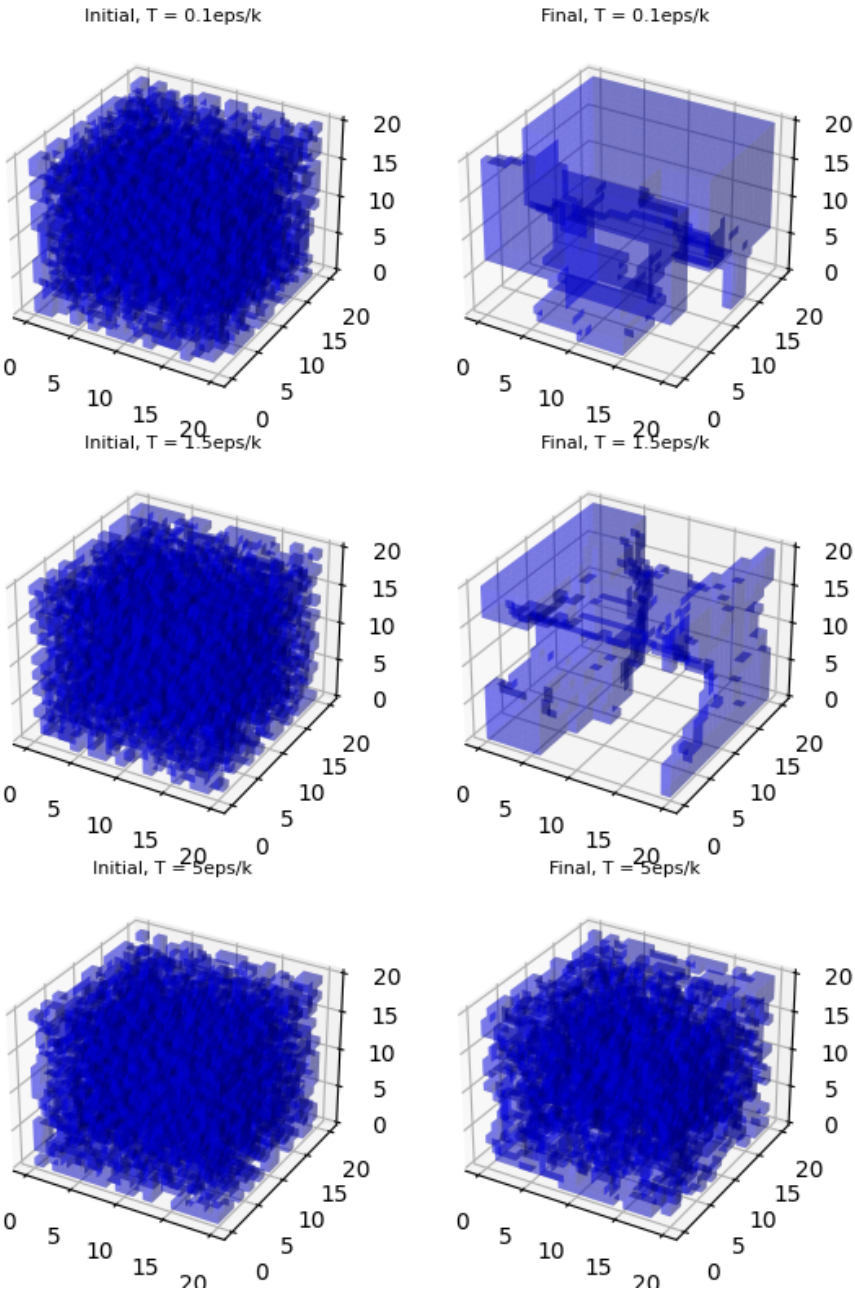

Figure 10:

Here I am doing a voxel plot with a light opacity, so that we may observe the overall structure of the system (rather than the apparent surface). Only the up spins are plotted - the abscence of a voxel implies down spin. On the left of figure 10, we see each state starts in an adequately random initial state. For the low-temperature test, it crystallizes to very few large chunks of similarly aligned dipoles. Similarly at the medium temperature, we see a connected structure forming. At the high temperature, the groups of similarly aligned dipoles are perhaps larger, but difficult to observe any particular order.

Hence, from the above, it appears that the simulation is working as designed.

3.3 Quantitative Analysis

Now that we've gained an intuitive sense for the model, and believe it is working, we look for a more rigorous method to confirm. In particular, we shall compute results for internal energy per dipole, free energy per dipole, entropy per dipole, specific heat capacity and mean absolute magnetism.

Although analytical solutions for the 3D Ising model do exist [1] [2], they are difficult to parse and work with. Instead, we compare our results to others who have approached the problem computationally, such as Sonsin et al. [3]. Although they do not compute the internal and free energy, specific heat capacity or entropy (and so we must reason ourselves whether or not our results are realistic), they do display both the mean magnetism, and an approach for determining the critical temperature.

The magnetic susceptibility χ is given by

$$\chi = \beta \left(\langle M^2 \rangle - \langle M \rangle^2 \right) \tag{1}$$

where M is the magnetism [3]. It is similar to the specific heat capacity, but for the magnetism rather than the temperature. From [3], it is "the greatness featuring a magnetic material according to their response to an applied magnetic field. The determination of χ can assist in the identification of phase transitions." Therefore, will we also compute and display χ for the 3D simulation to compare.

First, the quantities not looked at in the paper: internal energy per dipole, free energy per dipole, entropy per dipole and specific heat capacity per dipole. To compute and plot, we use the following script,

```
import simulation
import matplotlib.pyplot as plt
import numpy as np
import theory_quantities as tq

temperatures = np.linspace(0.25, 6, 10)
quantities = ["energy", "helmholtz", "entropy", "shc"]
quantity_titles = ["Internal Energy per Dipole", "Free Energy per Dipole", "Entropy per Dipole", "Specific
ylabels = ["Internal Energy [J]", "Free Energy [J]", "Entropy [J/K]", "Specific Heat Capacity [J/kg]"]

fig, axs = plt.subplots(4, 1)
fig.set_size_inches(6, 13, forward=True)

Ns = [5, 10, 20]
colours = ["red", "green", "blue"]
xshift = [-0.1, 0, 0.1]

for i, N in enumerate(Ns):
    # simulate + plot experimental val's
    for T in temperatures:
        print("N = " + str(N) +", T = "+str(T / max(temperatures)))
        # init simulation
        sim = simulation.ThreeDSimulation(N, T = T, eps = 1, k = 1)

        # run simulation to "equilibrium"
        sim.step(200000)

        # measure and plot quantities
        for ind, quantity in enumerate(quantities):
            mean, err = sim.time_average(quantity)
            mean = abs(mean)

            if T == min(temperatures):
                axs[ind].errorbar(T + xshift[i], mean, yerr=err, color=colours[i], linestyle = '', marker =
            else:
                axs[ind].errorbar(T + xshift[i], mean, yerr=err, color=colours[i], linestyle = '', marker =

# add titles/axis labels
for ind, quantity_title in enumerate(quantity_titles):
    axs[ind].set_title(quantity_title)
    axs[ind].set_xlabel("Temperature [eps/k]")
    axs[ind].set_ylabel(ylabels[ind])

    axs[ind].grid()
    axs[ind].legend()

axs[3].set_ylim(0, 0.1)

fig.tight_layout()

fig.savefig("assets/q3_ours.png")
```

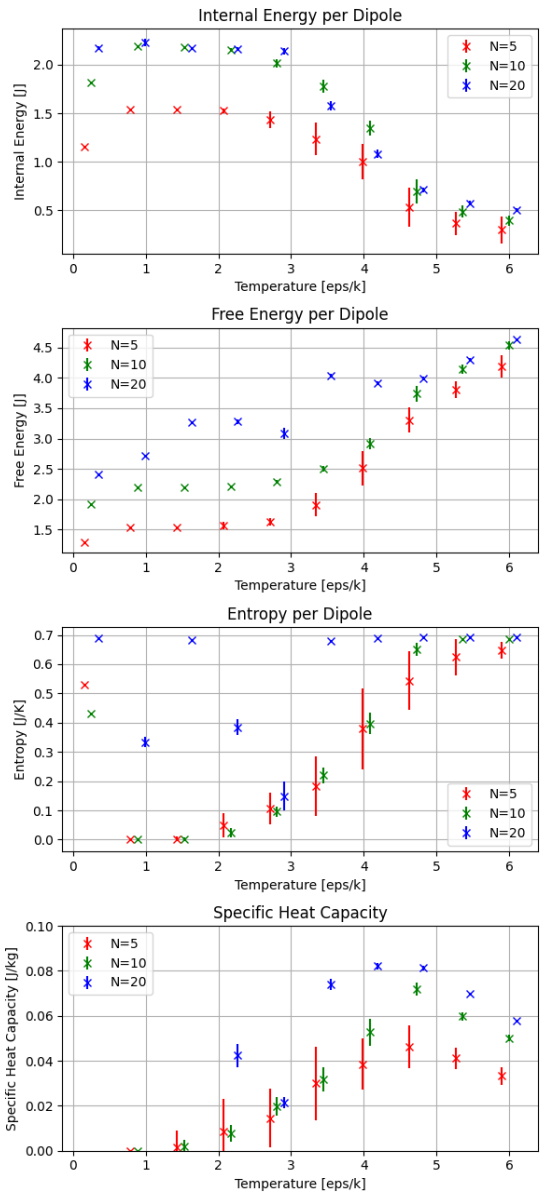

Figure 11: Internal and free energy, entropy, and specific heat capacity for 3D lattice of dipoles at varying temperatures and lattice sizes.

We observe very similar results to the 2D lattice observed earlier. Trends between each of the different dipole counts are also similar. Very few dipoles are used in the interest of saving time in running the simulation, however the trend of doubling the dipole count (in each direction - there are 64 times more dipoles in the $N = 20$ as compared to $N = 5$) shows how the trends change approaching an infinitely large 3D lattice.

Now, the quantities discussed in [3]: the mean absolute magnetism and magnetic susceptibility. To compute and plot, we use the following script,

```
import simulation
import matplotlib.pyplot as plt
import numpy as np
import theory_quantities as tq

temperatures = np.linspace(3, 8, 30)
quantities = ["magnet", "sus"]
quantity_titles = ["Mean Abs. Magnetism per Dipole", "Magnetic Susceptibility"]
ylabels = ["Magnetization", "Magnetic Susceptibility"]

fig, axs = plt.subplots(2, 1)
fig.set_size_inches(6, 13, forward=True)

Ns = [5, 10, 20]
colours = ["red", "green", "blue"]
xshift = [-0.1, 0, 0.1]
```

```python

for i, N in enumerate(Ns):
    # simulate + plot experimental val's
    for T in temperatures:
        print("N = " + str(N) +", T = "+str(T / max(temperatures)))
        # init simulation
        sim = simulation.ThreeDSimulation(N, T = T, eps = 1, k = 1)

        # run simulation to "equilibrium"
        sim.step(1000000)

        # measure and plot quantities
        for ind, quantity in enumerate(quantities):
            mean, err = sim.time_average(quantity)
            mean = abs(mean)

            if T == min(temperatures):
                axs[ind].errorbar(T + xshift[i], mean, yerr=err, color=colours[i], linestyle = '', marker =
            else:
                axs[ind].errorbar(T + xshift[i], mean, yerr=err, color=colours[i], linestyle = '', marker =

# add titles/axis labels
for ind, quantity_title in enumerate(quantity_titles):
    axs[ind].set_title(quantity_title)
    axs[ind].set_xlabel("Temperature [eps/k]")
    axs[ind].set_ylabel(ylabels[ind])

    axs[ind].grid()
    axs[ind].legend()

fig.tight_layout()

fig.savefig("assets/q3_theirs.png")
```

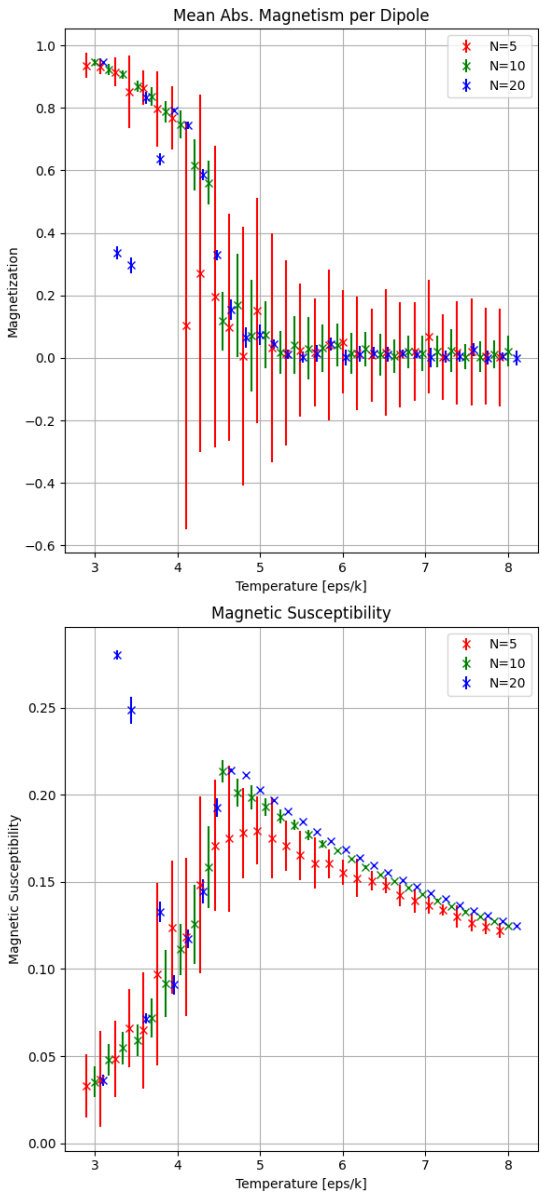

Figure 12: Magnetism quantities (abs. mean magnetism and magnetic susceptibility as defined in [3]) to compare against literature.

We observe extremely similar trends in figure 12 as to the similar plots in [3], for the plots with small N - unfortunately I do not have time to run long simulations (definitely not as long as in [3]), and so the number of times each dipole in anything past $N = 20$ is simulated is too low for it to reach some equilibrium reliably, hence the few stray results.

The magnetic susceptibility shows a sharp maximum at around $T = 4.8\epsilon/k$ - this is the expected critical temperature, and this result is in agreeance with [3]. Comparing this temperature to the earlier plots (both the magnetism in figure 12 and all quantities in 11), we see that $T_C \approx 4.8\epsilon/k$ is the temperature at which specific heat capacity reaches a maximum, magnetism bifurcates, and other quantities exhibit significant changes, further reinforcing the prediction that $T_C \approx 4.8\epsilon/k$ *is* the critical temperature.

3.4 Conclusion

We have successfully implemented, visualised and verified (against both reason and literature) a 3D implemention of the Ising model using Monte Carlo/Metropolis algorithm. Additionally, we have replicated a method for determining the critical temperature of a system by the magnetic susceptibility, and demonstrated it's accuracy.

References

[1] F. Canfora, "Kallen-lehman approach to 3d ising model," *Phsics Letters B*, vol. 646, 2007.

[2] D. Zhang, "Exact solution for three-dimensional ising model,"

[3] A. F. Sonsin, M. R. Cortes, D. R. Nunes, J. V. Gomes, and R. S. Costa, "Computational analysis of 3d ising model using metropolis algorithms," *Journal Physics Conference*, vol. 630, 2015.

4 Appendix

4.1 Simulation Class Code

```python
import numpy as np
import math
import random

class Simulation():
    def __init__(self, eps = 1, T = 1, k = 1):
        self.eps = eps
        self.k = k
        self.temperature = T * self.eps / self.k

        self.beta = 1 / (self.k * self.temperature)

    def get_spins(self):
        return self.spins

    def print_chance_summary(self):
        chances = self.flip_chances.flatten()
        minChance = min(chances)
        meanChance = np.mean(chances)

        print("The simulation had at least", minChance, "chances for each dipole to flip.")
        print("On average, there were", meanChance, "flips.")

    def get_multiplicity(self):
        unique, counts = np.unique(self.spins.flatten(), return_counts=True)
        comb = dict(zip(unique, counts))
        if 1 in comb.keys():
            Nup = comb[1]
        else:
            Nup = self.N - comb[-1]

        return math.comb(self.N, Nup)


    def get_energy(self):
        pass

    def get_energy_per_dipole(self):
        total_energy = self.get_energy()
        return total_energy / self.N

    def get_helmholtz_per_dipole(self):
        S = self.k * math.log(self.get_multiplicity())
        F = self.get_energy() - self.temperature * S

        return F / self.N

    def get_entropy_per_dipole(self):
        entropy = self.k * math.log(self.get_multiplicity())

        return entropy / self.N

    def get_reduced_magnetism_per_dipole(self):
        mu_b = 1                # replace with actual val later
        sbar = np.mean(self.spins)
        M = mu_b * self.N * sbar

        return M / (mu_b * self.N)

    def get_magnetic_susceptibility(self):
        mu_b = 1
```

```
        Msq = np.mean(np.power(self.spins, 2))
        M = self.get_reduced_magnetism_per_dipole()

        return self.beta * (Msq - M**2)

    def get_specific_heat_capacity(self):
        pass

    def time_average(self, quantity):
        if quantity == "entropy":
            func = self.get_entropy_per_dipole
        elif quantity == "energy":
            func = self.get_energy_per_dipole
        elif quantity == "helmholtz":
            func = self.get_helmholtz_per_dipole
        elif quantity == "magnet":
            func = self.get_reduced_magnetism_per_dipole
        elif quantity == "shc":
            func = self.get_specific_heat_capacity
        elif quantity == "sus":
            func = self.get_magnetic_susceptibility
        else:
            raise Exception("Unknown quantity \"" + quantity + "\" for time average!")

        vals = np.zeros(20)
        for i in range(len(vals)):
            self.step(steps=5000)
            vals[i] = func()

        mean = np.mean(vals)
        err = np.std(vals)

        return mean, err

class OneDSimulation(Simulation):
    def __init__(self, dipole_count, eps = 1, T = 1, k = 1):
        self.spins = np.zeros(dipole_count)
        for i in range(dipole_count):
            self.spins[i] = random.choice([-1, 1])

        self.flip_counts = np.zeros(dipole_count)
        self.flip_chances = np.zeros(dipole_count)

        self.N = dipole_count
        self.width = dipole_count

        super().__init__(eps, T, k)

    def get_energy(self):
        temp_sum = 0
        for i in range(len(self.spins) - 1): # every particle except the last
            temp_sum += self.spins[i] * self.spins[i + 1]
        # and the last looking at the first
        temp_sum += self.spins[-1] * self.spins[0]

        U = temp_sum * (- self.eps)

        return U

    def local_energy_difference(self, index):
        # only need to look at change about the index
        if index > 0 and index < len(self.spins) - 1:
            sum_init = self.spins[index] * (self.spins[index - 1] + self.spins[index + 1])
        elif index == 0:
```

```
            sum_init = self.spins[index] * (self.spins[1] + self.spins[-1])
        elif index == len(self.spins) - 1:
            sum_init = self.spins[index] * (self.spins[0] + self.spins[-2])
        else:
            raise Exception("I don't know how to deal with this spin index for local_energy_difference")

        return 2 * self.eps * sum_init

    def step(self, steps=200000):
        j = 0
        rands = np.random.randint(self.width, size=steps)
        while j < steps:
            i = rands[j]
            j += 1

            self.flip_chances[i] += 1

            dU = self.local_energy_difference(i)

            if dU <= 0:
                self.spins[i] *= -1
                self.flip_counts[i] += 1
            else:
                prob = np.exp(- self.beta * dU)
                if random.random() < prob:
                    self.spins[i] *= -1
                    self.flip_counts[i] += 1

    def get_specific_heat_capacity(self):
        # C = <U^2> - <U>^2 / kT^2
        U = self.get_energy()

        temp_sum = 0
        for i in range(len(self.spins) - 1): # every particle except the last
            temp_sum += (self.spins[i] * self.spins[i + 1])**2
        # and the last looking at the first
        temp_sum += (self.spins[-1] * self.spins[0])**2

        U2 = temp_sum * (- self.eps)

        C = (U2 - U)/(self.k * self.temperature**2)

        return -C / self.N

class TwoDSimulation(Simulation):
    def __init__(self, dipole_count, eps = 1, T = 1, k = 1.38 * 1e-23):
        self.spins = np.zeros((dipole_count, dipole_count))
        for i in range(dipole_count):
            for j in range(dipole_count):
                self.spins[i, j] = random.choice([-1, 1])

        self.flip_counts = np.zeros((dipole_count, dipole_count))
        self.flip_chances = np.zeros((dipole_count, dipole_count))

        self.width = dipole_count
        self.N = dipole_count**2

        super().__init__(eps, T, k)

    def get_energy(self):
        temp_sum = 0

        for i in range(self.width - 1):
```

```
            for j in range(self.width - 1):
                x = i + 1
                y = j + 1

                if i == self.width:
                    x = 0
                if j == self.width:
                    y = 0

                temp_sum += self.spins[i, j] * self.spins[x, j]
                temp_sum += self.spins[i, j] * self.spins[i, y]

        U = temp_sum * (- self.eps)

        return U

    def local_energy_difference(self, x, y):
        # only need to look at change about the index

        left = self.spins[self.width - 1 if x == 0 else x - 1, y]
        right = self.spins[0 if x == self.width - 1 else x + 1, y]
        top = self.spins[x, self.width - 1 if y == 0 else y - 1]
        bottom = self.spins[x, 0 if y == self.width - 1 else y + 1]

        return 2 * self.spins[x, y] * (top + bottom + left + right)

    def step(self, steps=1000):
        j = 0
        rands = np.random.randint(self.width, size=(2, steps))
        while j < steps:
            x = rands[0, j]
            y = rands[1, j]

            j += 1

            self.flip_chances[x, y] += 1

            dU = self.local_energy_difference(x, y)

            if dU <= 0:
                self.spins[x, y] *= -1
                self.flip_counts[x, y] += 1
            else:
                prob = np.exp(- self.beta * dU)
                if random.random() < prob:
                    self.spins[x, y] *= -1
                    self.flip_counts[x, y] += 1

    def get_specific_heat_capacity(self):
        # C = <U^2> - <U>^2 / kT^2
        U = self.get_energy()

        temp_sum = 0
        for i in range(self.width - 1):
            for j in range(self.width - 1):
                x = i + 1
                y = j + 1

                if i == self.width:
                    x = 0
                if j == self.width:
                    y = 0

                temp_sum += (self.spins[i, j] * self.spins[x, j])**2
```

```
                temp_sum += (self.spins[i, j] * self.spins[i, y])**2

        U2 = temp_sum * (- self.eps)

        C = (U2 - U)/(self.k * self.temperature**2)

        return -C / self.N

    def set_temp(self, T):
        self.temperature = T * self.eps / self.k
        self.beta = 1 / (self.k * self.temperature)

class ThreeDSimulation(Simulation):
    def __init__(self, dipole_count, eps = 1, T = 1, k = 1.38 * 1e-23):
        self.spins = np.zeros((dipole_count, dipole_count, dipole_count))
        for x in range(dipole_count):
            for y in range(dipole_count):
                for z in range(dipole_count):
                    self.spins[x, y, z] = random.choice([-1, 1])

        self.flip_counts = np.zeros((dipole_count, dipole_count, dipole_count))
        self.flip_chances = np.zeros((dipole_count, dipole_count, dipole_count))

        self.width = dipole_count
        self.N = dipole_count**3 # assume cube lattice

        super().__init__(eps, T, k)

    def get_energy(self):
        temp_sum = 0

        for i in range(self.width - 1):
            for j in range(self.width - 1):
                for k in range(self.width - 1):
                    x = i + 1
                    y = j + 1
                    z = k + 1

                    if i == self.width:
                        x = 0
                    if j == self.width:
                        y = 0
                    if k == self.width:
                        z = 0

                    temp_sum += self.spins[i, j, k] * self.spins[x, j, k]
                    temp_sum += self.spins[i, j, k] * self.spins[i, y, k]
                    temp_sum += self.spins[i, j, k] * self.spins[i, j, z]

        U = temp_sum * (- self.eps)

        return U

    def local_energy_difference(self, x, y, z):
        # only need to look at change about the index

        front = self.spins[self.width - 1 if x == 0 else x - 1, y, z]
        behind = self.spins[0 if x == self.width - 1 else x + 1, y, z]

        left = self.spins[x, self.width - 1 if y == 0 else y - 1, z]
        right = self.spins[x, 0 if y == self.width - 1 else y + 1, z]

        above = self.spins[x, y, self.width - 1 if z == 0 else z - 1]
        below = self.spins[x, y, 0 if z == self.width - 1 else z + 1]
```

```python

        return 2 * self.spins[x, y, z] * (front + behind + left + right + above + below)

    def step(self, steps=1000):
        j = 0
        rands = np.random.randint(self.width, size=(3, steps))
        while j < steps:
            x = rands[0, j]
            y = rands[1, j]
            z = rands[2, j]

            j += 1

            self.flip_chances[x, y, z] += 1

            dU = self.local_energy_difference(x, y, z)

            if dU <= 0:
                self.spins[x, y, z] *= -1
                self.flip_counts[x, y, z] += 1
            else:
                prob = np.exp(- self.beta * dU)
                if random.random() < prob:
                    self.spins[x, y, z] *= -1
                    self.flip_counts[x, y, z] += 1

    def get_specific_heat_capacity(self):
        # C = <U^2> - <U>^2 / kT^2
        U = self.get_energy()

        temp_sum = 0
        for i in range(self.width - 1):
            for j in range(self.width - 1):
                for k in range(self.width - 1):
                    x = i + 1
                    y = j + 1
                    z = k + 1

                    if i == self.width:
                        x = 0
                    if j == self.width:
                        y = 0
                    if k == self.width:
                        z = 0

                    temp_sum += (self.spins[i, j, k] * self.spins[x, j, k])**2
                    temp_sum += (self.spins[i, j, k] * self.spins[i, y, k])**2
                    temp_sum += (self.spins[i, j, k] * self.spins[i, j, z])**2

        U2 = temp_sum * (- self.eps)

        C = (U2 - U)/(self.k * self.temperature**2)

        return -C / self.N
```

4.2 Theoretical Quantities Code

```python
import numpy as np
import math

# k = 1.38 * 1e-23 # Boltzmann's
k = 1 # Boltzmann's

def u(beta, eps, T):
    return -eps * np.tanh(beta * eps)

def f(beta, eps, T):
    exponential = np.exp(-2 * eps * beta)
    log_term = np.log(1 + exponential)

    return -eps - k * T * log_term

def S(beta, eps, T):
    lhs = (eps / T) * (1 - np.tanh(beta * eps))
    rhs = k * math.log(1 + np.exp(-2 * eps * beta))

    return lhs + rhs

def c(beta, eps, T):
    numerator = eps**2 * beta
    denominator = T * (np.cosh(beta * eps))**2

    return numerator / denominator
```

4.3 Q1A Code

```python
import simulation
import matplotlib.pyplot as plt

print("=== Question 1 A ===")

fig, axs = plt.subplots(3, 1)
fig.tight_layout()
fig.subplots_adjust(top=0.95)
fig.subplots_adjust(right=0.95)
fig.set_size_inches(6, 6, forward=True)

for ind, temp in enumerate([2, 1, 0.5]):
    # init simulation
    sim = simulation.OneDSimulation(100, T = temp)

    # run sim
    sim.step(steps = 100 * 1000 * 2)

    # plot simulation
    x = 0
    y = ind

    axs[y].imshow(sim.get_spins().reshape(1,-1), aspect="auto", cmap="binary")
    axs[y].set_title("Temperature " + str(temp) + "eps/k", fontsize=15)

    axs[y].get_xaxis().set_visible(False)
    axs[y].get_yaxis().set_visible(False)



plt.savefig("assets/q1a.png")
```

4.4 Q1C Code

```python
import simulation
import matplotlib.pyplot as plt
import numpy as np
import theory_quantities as tq

print("=== Question 1 C ===")
temperatures = np.linspace(0.01, 6, 25)
quantities = ["energy", "helmholtz", "entropy", "shc","magnet"]
quantity_titles = ["Internal Energy per Dipole", "Free Energy per Dipole", "Entropy per Dipole", "Specific Heat Capacity", "Reduced Magnetism Per Dipole"]
ylabels = ["Internal Energy [J]", "Free Energy [J]", "Entropy [J/K]", "Specific Heat Capacity [J/kg]", "Reduced Magnetism [T]"]

fig, axs = plt.subplots(5, 1)
fig.set_size_inches(6, 15, forward=True)

# simulate + plot experimental val's
for T in temperatures:
    print(T)
    # init simulation
    sim = simulation.OneDSimulation(100, T = T, eps = 1, k = 1)

    # run simulation to "equilibrium"
    sim.step(100 * 1000 * 2)

    # measure and plot quantities
    for ind, quantity in enumerate(quantities):
        mean, err = sim.time_average(quantity)

        if T == min(temperatures):
            axs[ind].errorbar(T, mean, yerr=err, color="red", linestyle = '', marker = "x", label="Experimental")
        else:
            axs[ind].errorbar(T, mean, yerr=err, color="red", linestyle = '', marker = "x")

# calc + plot theory val's
temperatures = np.linspace(0.01, 6, 250)
theory_funcs = [tq.u, tq.f, tq.S, tq.c]
for i in range(4):
    y = np.zeros(len(temperatures))
    for j in range(len(temperatures)):
        beta = 1 / temperatures[j]
        y[j] = theory_funcs[i](beta, 1, temperatures[j])

    axs[i].plot(temperatures, y, color="blue", label="Theoretical")

# add titles/axis labels
for ind, quantity_title in enumerate(quantity_titles):
    axs[ind].set_title(quantity_title)
    axs[ind].set_xlabel("Temperature [eps/k]")
    axs[ind].set_ylabel(ylabels[ind])

    axs[ind].grid()
    axs[ind].legend()

fig.tight_layout()

fig.savefig("assets/q1c.png")
```

4.5 Q1E Code

```python
print("=== Question 1 E ===")

import simulation
import numpy as np
import matplotlib.pyplot as plt


temperatures = [0.5, 1.0, 2.0]
dipoles = [100, 500]

for N in dipoles:
    fig, axs = plt.subplots(3, 1)
    fig.set_size_inches(4, 8, forward=True)

    for ind, T in enumerate(temperatures):
        print("T =",T)
        m_values = np.zeros(200)
        for i in range(len(m_values)):
            if i % 20 == 0:
                print(i / len(m_values))
            # initialise new simulation
            sim = simulation.OneDSimulation(N, T = T)
            # step until (hopeful) equilibrium
            sim.step(N * 1000)
            # record + store time average of reduced
            # magnetism per dipole
            m, _ = sim.time_average("magnet")
            m_values[i] = m

        # plot histogram
        axs[ind].hist(m_values, color="xkcd:blue")
        # fancy plot accoutrement
        axs[ind].set_xlim(-1, 1)
        axs[ind].set_title("Temperature " + str(T) + "eps/k", fontsize=15)
        axs[ind].set_xlabel("Reduced magnetism per dipole")
        axs[ind].grid()

    fig.tight_layout()
    fig.savefig("assets/q1e_" + str(N) + "dipoles.png")
```

4.6 Q2A Code

```python
import simulation
import matplotlib.pyplot as plt

print("=== Question 2 A ===")

fig, axs = plt.subplots(3, 2)
fig.set_size_inches(6, 8, forward=True)

# for temperature = 3, 2, 1 eps/k
for ind, temp in enumerate([3, 2, 1]):
    print("T =", temp)
    # init simulation
    sim = simulation.TwoDSimulation(100, T = temp)

    # plot initial state
    axs[ind, 0].imshow(sim.get_spins(), aspect=1, cmap="binary")
    axs[ind, 0].set_title("Initial, T = " + str(temp) + "eps/k", fontsize=15)

    axs[ind, 0].get_xaxis().set_visible(False)
    axs[ind, 0].get_yaxis().set_visible(False)

    # run sim until equilibrium
    sim.step(steps = 100 * 100 * 1000)

    # plot final state
    axs[ind, 1].imshow(sim.get_spins(), aspect=1, cmap="binary")
    axs[ind, 1].set_title("Final, T = " + str(temp) + "eps/k", fontsize=15)

    axs[ind, 1].get_xaxis().set_visible(False)
    axs[ind, 1].get_yaxis().set_visible(False)

# tidy layout and save
fig.tight_layout()
plt.savefig("assets/q2a.png")
```

4.7 Q2B Code

```python
import numpy as np
import matplotlib.pyplot as plt
import simulation

temperatures = np.linspace(1.7, 2.6, 10)

fig, axs = plt.subplots(len(temperatures) // 2, 2)
fig.set_size_inches(4, 10, forward=True)

dipoles = 50

for x, T in enumerate(temperatures):
    print(x / (len(temperatures) - 1))
    sim = simulation.TwoDSimulation(dipoles, T = T)

    sim.step(dipoles**2 * 2000)

    y = int(np.floor(x / 5))
    x = x % 5

    axs[x,y].imshow(sim.get_spins(), aspect=1, cmap="binary")
    axs[x,y].set_title("T = " + str(round(T, 2)) + "eps/k", fontsize=10)

    axs[x,y].get_xaxis().set_visible(False)
    axs[x,y].get_yaxis().set_visible(False)

fig.tight_layout()
plt.savefig("assets/q2b.png")
```

4.8 Q2C Code

```python
import simulation
import matplotlib.pyplot as plt
import numpy as np
import theory_quantities as tq

print("=== Question 2 C ===")
temperatures = np.linspace(0.25, 6, 10)
quantities = ["energy", "helmholtz", "entropy", "shc","magnet"]
quantity_titles = ["Internal Energy per Dipole", "Free Energy per Dipole", "Entropy per Dipole", "Specific Heat Capacity", "Reduced Magnetism Per Dipole"]
ylabels = ["Internal Energy [J]", "Free Energy [J]", "Entropy [J/K]", "Specific Heat Capacity [J/kg]", "Abs. Reduced Magnetism [T]"]

fig, axs = plt.subplots(5, 1)
fig.set_size_inches(6, 15, forward=True)

Ns = [20, 50, 100]
colours = ["red", "green", "blue"]
xshift = [-0.1, 0, 0.1]

for i, N in enumerate(Ns):
    # simulate + plot experimental val's
    for T in temperatures:
        print("N = " + str(N) +", T = "+str(T / max(temperatures)))
        # init simulation
        sim = simulation.TwoDSimulation(N, T = T, eps = 1, k = 1)

        # run simulation to "equilibrium"
        sim.step(N * N * 10000)

        # measure and plot quantities
        for ind, quantity in enumerate(quantities):
            mean, err = sim.time_average(quantity)
            mean = abs(mean)

            if T == min(temperatures):
                axs[ind].errorbar(T + xshift[i], mean, yerr=err, color=colours[i], linestyle = '', marker = "x", label="N="+str(N))
            else:
                axs[ind].errorbar(T + xshift[i], mean, yerr=err, color=colours[i], linestyle = '', marker = "x")

# add titles/axis labels
for ind, quantity_title in enumerate(quantity_titles):
    axs[ind].set_title(quantity_title)
    axs[ind].set_xlabel("Temperature [eps/k]")
    axs[ind].set_ylabel(ylabels[ind])

    axs[ind].grid()
    axs[ind].legend()

fig.tight_layout()

fig.savefig("assets/q2c.png")
```

4.9 Q2D Code

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

import simulation

def pos(lst):
    return [x for x in lst if x > 0] or None
def neg(lst):
    return [x for x in lst if x < 0] or None

fig, ax = plt.subplots()
fig.set_size_inches(6, 6, forward=True)

colours = ["red", "green", "blue"]
xshift = [-0.1, 0, 0.1]

Ns = [5, 10, 20]
for i, N in enumerate(Ns):
    temperatures = np.linspace(0.05, 6, 10)

    pos_means = []
    neg_means = []
    dpos_means = []
    dneg_means = []
    for t in temperatures:
        print("N="+str(N)+", T="+str(t / max(temperatures)))
        temp_means = np.zeros(100)
        for j in range(len(temp_means)):
            # initialise simulation
            sim = simulation.TwoDSimulation(N, T = t)
            # step until equilibrium
            sim.step(N**2 * 500)
            # measure magnetism
            mean, _ = sim.time_average("magnet")
            temp_means[j] = mean
        pos_means.append(np.mean(pos(temp_means)))
        neg_means.append(np.mean(neg(temp_means)))
        dpos_means.append(np.std(pos(temp_means)))
        dneg_means.append(np.std(neg(temp_means)))

    ax.errorbar(temperatures + xshift[i], pos_means, yerr=dpos_means, color=colours[i], linestyle = ''
    ax.errorbar(temperatures + xshift[i], neg_means, yerr=dneg_means, color=colours[i], linestyle = ''

ax.grid()
ax.legend()
ax.set_xlabel("Temperature [eps/k]")
ax.set_ylabel("Avg. Neg. Magnetism            Avg. Pos. Magnetism")
ax.set_ylim(-1.05, 1.05)
ax.set_title("Mean positive and negative magnetism,\n varying with temperature and dipole count")
fig.savefig("assets/q2d.png")
```

4.10 Q2E Code

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

import simulation

fig, axs = plt.subplots(3, 1)
fig.set_size_inches(6, 9, forward=True)

N = 25

# initialise "cold" simulation
print("init")
T = 1
sim = simulation.TwoDSimulation(N, T = T)

# step to equilibrium
sim.step(N**2 * 1000)

# display
axs[0].imshow(sim.get_spins(), aspect=1, cmap="binary")
axs[0].set_title("Initial, T = 1eps/k", fontsize=10)

# slowly heat to T = 3
# do this by increasing temperature slightly, stepping 10 times per dipole, repeat
print("heating")
while T < 3:
    T += 0.1
    sim.set_temp(T)
    sim.step(N**2 * 10)


# display
axs[1].imshow(sim.get_spins(), aspect=1, cmap="binary")
axs[1].set_title("Heated, T = 3eps/k", fontsize=10)

# cool to T = 1
# same process as before
print("cooling")
while T > 1:
    T -= 0.1
    sim.set_temp(T)
    sim.step(N**2 * 10)
# reach an equilibrium at cooled state
sim.step(N**2 * 100)

# display
axs[2].imshow(sim.get_spins(), aspect=1, cmap="binary")
axs[2].set_title("Cooled, T = 1eps/k", fontsize=10)

# remove axis
for i in range(3):
    axs[i].get_xaxis().set_visible(False)
    axs[i].get_yaxis().set_visible(False)

fig.tight_layout()
fig.savefig("assets/q2e.png")
```