A MONTE CARLO SIMULATION OF THE 2D ISING MODEL

The results of a Monte Carlo simulation of the nearest-neighbour Ising model using the Metropolis algorithm are presented. In the absence of an external field, the magnetisation, susceptibility and heat capacity were all found to vary significantly near the critical temperature, where there was found to be a second order phase transition. These sharp transitions when considered with the finite size scaling of the system were used to produce an estimate the critical temperature in the thermodynamic limit of an infinitely large lattice, giving $T_c(N = \infty) = 2.274 \pm 0.005$ [J/k_B]. An estimation of the critical exponent β was found from the magnetisation as $\beta = 0.133 \pm 0.008$ [J^{-1}]. Both of these values were consistent with the theoretical values predicted by Onsager of $T_c = 2/(1 + \ln\sqrt{2})$ and $\beta = 1/8$. When an external field was applied, there was found to be a non-zero magnetisation above the critical temperature, and only one equilibrium state below it. Finally, the phenomenon of hysteresis is demonstrated for temperatures below the critical temperature.

1 INTRODUCTION

The Ising model is a simple model used to represent ferromagnetic and anti-ferromagnetic materials. The model involves the creation of a square lattice of atoms which can each have an associated quantum mechanical spin of +1 (spin up) or -1 (spin down). These spins each give rise to dipole moments within the atoms, and can cause measureable macroscopic effects when aligned in the same due to the superposition of their magnetic fields.
The total energy (Hamiltonian) of the system predicted by the Ising model for an NxN lattice is given by:

$$E = -J \sum_{\langle i,j \rangle} s_i s_j - \mu H \sum_{i=1}^{N^2} s_i \qquad (1)$$

The first term refers to a sum over the product of each spin with that of its nearest neighbours. The coupling parameter, J, is the interaction energy between the spins and its sign determines whether the spins favour alignment or anti-alignment (i.e. $J > 0$ for ferromagnetism or $J < 0$ for anti-ferromagnetism). The remaining terms, μ and H refer to the magnetic moment per spin and external field strength respectively.
It turns out that below a specific temperature, the material behaves as a ferromagnet where spins tend to align in a preferred direction and thus induce a net magnetisation, even in the absence of an external field. Above this temperature the thermal fluctuations in the spins dominate, causing the spins to be randomly oriented and therefore no net magnetisation; this corresponds to a paramagnet in which a net magnetisation is only induced upon the application of an external field. This specific temperature is called the critical temperature and was derived by Lars Onsager in 1944 as $T_c = 2/(1 + \ln\sqrt{2})$ in the limit of an infinitely large lattice (where temperatures throughout are given in units of J/k_B). This behaviour is known as a second order phase transition with the magnetisation as the order parameter of the transition.

The following section provides an analysis of the computational aspects of the problem. Section 3 entails the implementation of the program, including the execution of the Metropolis algorithm (the algorithm used to enact the Ising model) and an outline of the methods used to optimise the program. Section 4 contains the results and a discussion of their meaning. Finally, the conclusions of this paper are presented in section 5.

2 ANALYSIS

In Physics the desire when analysing any physical system is always to find a closed form analytic solution to the problem. Unfortunately, for many problems analytic techniques are intractable, and for those that are, numerous approximations must be made.
In such cases that we look to numerical simulation techniques instead, with a particularly successful example being the Ising model.[1]
Consider a 32 x 32 lattice containing a total of 2^{10} ($\approx$1000) spin sites, each spin site has two degrees of freedom (spin up/down). The total number of permissible microstates is 2^{1000} ($\approx 10^{300}$) and are all equally likely. This is clearly far too many microstates to sum over in order to find the ensemble averages and thermodynamic variables in a computer simulation, therefore we take a different approach.
Instead of choosing configurations randomly, then weighting them with the Boltzmann factor, we choose configurations with probability $\exp(-E/k_B T)$ (E is the configuration energy and T is the reservoir

[1] There is in fact an analytical solution to the Ising model for 2 dimensions, but not for any higher dimensions.

temperature) then weight them evenly. This approach underpins the Metropolis algorithm (the algorithm used in this paper). The algorithm works as follows:

1) Create a system of spins with some pre-set orientations.
2) Select a random lattice site and find the energy required to flip the spin (ΔE) with equation (1).
   a) If $\Delta E < 0$, flip the spin.
   b) If $\Delta E > 0$ and $\exp(-E/k_B T) > \mathrm{p}$, where p is a uniform random number between 0 and 1, then flip the spin.
   c) Otherwise, leave the spin unchanged.
3) Repeat 2) for another N^2 randomly selected lattice sites.
4) Iterate steps 2) to 3) to evolve the system in 'time'.

In this algorithm evolution in 'time' corresponds to performing Monte Carlo sweeps over the lattice (steps 2) and 3) correspond to one sweep). A large number of these sweeps were performed to ensure the system had reached equilibrium.
To avoid the introduction of artefacts due to the sites at the lattice boundaries not having 4 neighbours, and to improve the quality of the simulation results, toroidal boundary conditions were introduced. To visualise this, imagine wrapping the left edge of the lattice back around to the right edge and the top to the bottom simultaneously such that each site then has the same number of neighbours.

3 IMPLEMENTATION

The source code (written in Python) may be found in appendix B. The simulation was scaled by setting the magnitude of J, k_B and μ to unity, however the sign of J could be changed to switch between ferromagnetism (J = +1) and anti-ferromagnetism (J = −1).
The lattice of spins was initialised by the one of the three *initialise* lambda functions, which generated either a random set of spins or a uniform lattice with all spins pointing up depending on the specific investigation and the temperature.
The function *spin_flip_energy* takes the spin at a specific lattice site, computes its nearest neighbours under the assertion of toroidal boundary conditions, then returns the energy required to flip the spin.
The main part of the Metropolis algorithm is executed by *state_change*, which performs steps 2) and 3) of the aforementioned algorithm. The total energy of the lattice and absolute magnetisation per site are calculated in the functions *find_magnetisation* and *energy* respectively.
From fluctuation-dissipation theorem, the standard deviations in magnetisation and energy are used to calculate the heat capacity (C), and magnetic susceptibility (χ):

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

$$\chi = \frac{\sigma_M^2}{k_B T} \qquad (3)$$

We find that heat capacity and magnetic susceptibility both diverge at critical temperature, thus they are good graphs from which to extract an estimate of T_c (this divergence becomes increasingly evident for larger N). This also means that there is a significant error associated with the data points close to this temperature, as such a filter was applied to these curves to smooth out some of the variations if the graph was being used to estimate the critical temperature.

3.1 PERFORMANCE

The primary area of optimisation was in the function *state_change* since it was the most frequently called function. There were two major changes to increase the efficiency of the code.
For each sweep through the lattice, instead of calling the random number generator three times in each iteration of the *for* loop to find a random lattice site and generate p (the random number for the Boltzmann factor), three arrays of random numbers were generated outside of the *for* loop then indexed into. On average this reduced the CPU time for any executable by a third.
The second major optimisation came from noting the cost of calling the exponential function N^2 times each sweep. Clearly the Boltzmann factor is redundant if $\Delta E < 0$ as the spin would flip anyway, hence its evaluation is only necessary if $\Delta E > 0$. There are only three unique scenarios in which a this can occur and they result in energy changes of 0J, 4J and 8J. By calculating the Boltzmann factor associated with these values once outside of the loop, then just calling the correct value using the function *boltzmann_factor* the CPU time was reduced by another third.
For temperatures close to T_c the system required more Monte Carlo sweeps to reach its approximate equilibrium state, and from the ensuing plots of the magnetisation autocorrelation, the number of sweeps thermodynamic quantities must be averaged over to get a representative result is also far higher around T_c. Thus, the system was permitted far more sweeps to equilibrate and average quantities over around the critical temperature.
As an indication of the programs speed, the CPU time required to perform 1000 MC sweeps for a 20 x 20 lattice was 4s, the same number of sweeps for a 64 x 64 lattice required 35s. The plot in figure 5 involved averaging over 50,000 sweeps for each of the 14 temperature points in the range [2.1, 2.5] and 10,000 sweeps for each of the 14 points outside of that range. This required a CPU time of 88 min.

4 RESULTS AND DISCUSSION

4.1 ZERO FIELD (H = 0)

For low temperatures ($T \ll T_c$), the lattice equilibrium was found to be with all spins pointing in the same direction (spin up or down). As the temperature approached T_c from below, the distribution of spins became less uniform as the thermal motion began to dominate. For temperatures below T_c the lattice was initialised uniformly (with all spins pointing up), and for temperatures around and above T_c the lattice was initialised with a random distribution of spins. The advantages of this were two-fold, firstly it reduced the number of sweeps required to reach the near equilibrium state, and secondly it helped to avoid the following issue: starting from an initial configuration of randomly oriented spins, approximately a quarter of all simulations would fall into a local minimum (appendix A), rather than the desired global minimum (figure 2). This occurred predominantly at low temperatures as the system didn't have the required thermal energy to overcome the energy barrier and reach the global minimum. Starting from an ordered state meant the acquisition of such a local minimum almost never occurred.

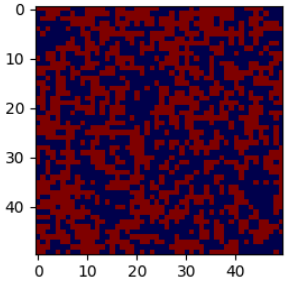

Figure 1: The spin orientations for a 50x50 lattice over 350 sweeps at T = 5 [J/k_B] for a positive interaction energy J, where blue is spin up and red is spin down. The spins are randomly aligned for $T > T_c$, with no spontaneous magnetisation.

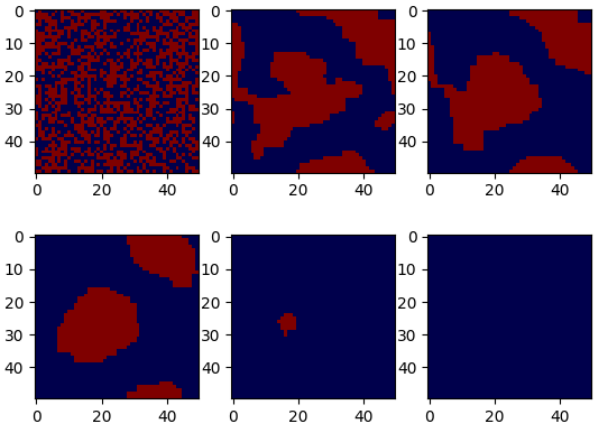

Figure 2: The spin orientations for a 50x50 lattice over 350 sweeps at T = 0.5 [J/k_B] for a positive interaction energy J, where blue is spin up and red is spin down. The spins all flip to align, demonstrating ferromagnetism.

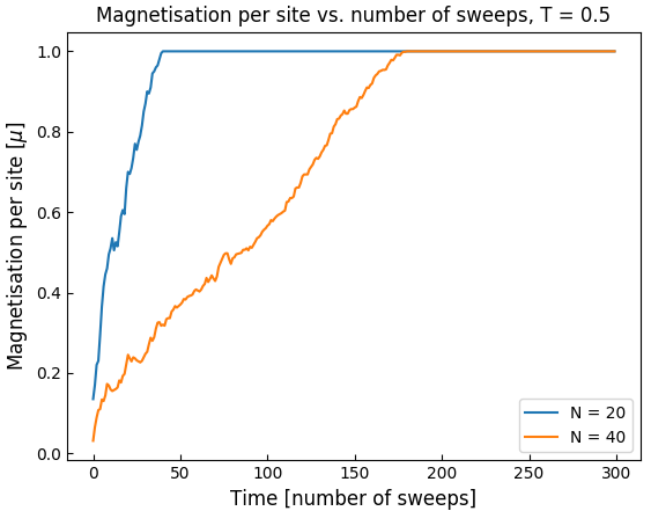

Figure 3: Magnetisation per site against the number of sweeps, illustrating the time required for the system to reach equilibrium at T = 0.5 for two different lattice sizes.

Plots such as figure 3 were used to provide a quantitative estimate for the number of sweeps required to achieve the near-equilibrium state before sampling any thermodynamic quantities.

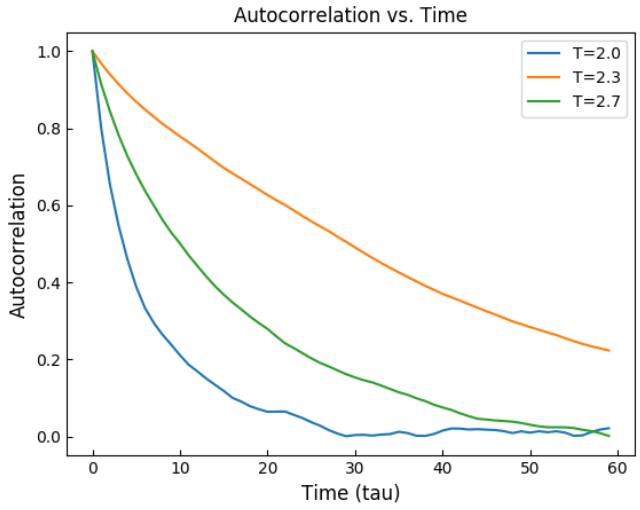

Figure 4: Magnetisation autocorrelation vs time once equilibrium has been reached. Illustrates how correlated previous sweep configurations are to the next. The autocorrelation decays exponentially with time.

Figures 4 and 5 were used to give an estimate of the number of sweeps required to get an accurate result when taking averages, for example for a lag time of 100, at least 500 sweeps would be averaged over.

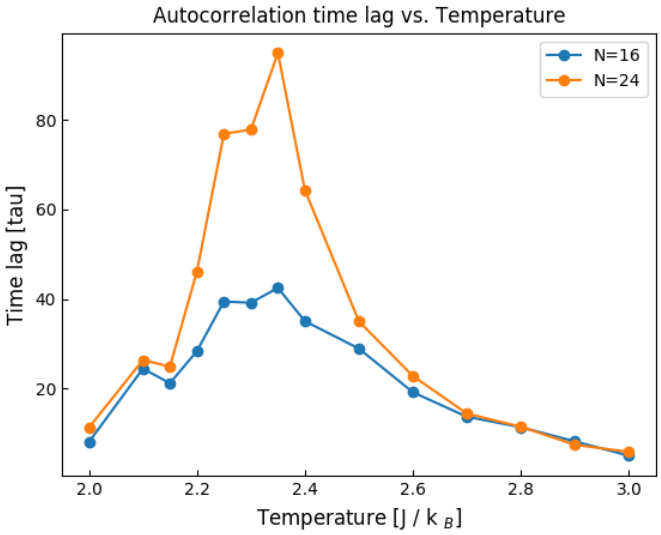

Figure 5: Autocorrelation time lag vs time once equilibrium has been reached (time lag is the time over which the autocorrelation falls to 1/e).

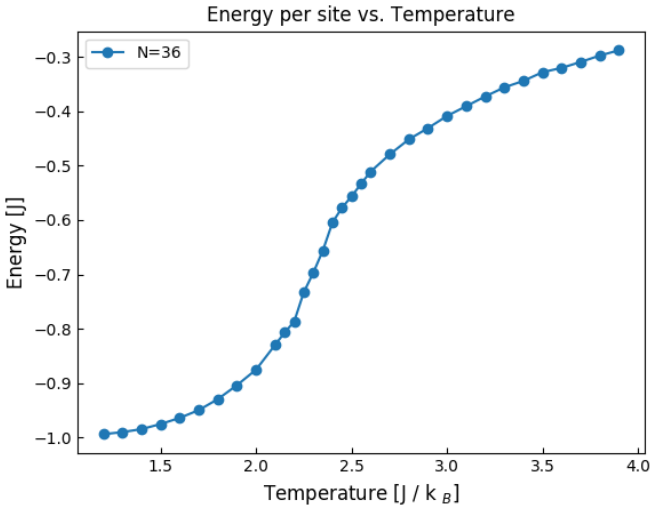

Figure 6: Energy per site vs temperature for a 36x36 lattice, shows a sharper increase in energy around the critical temperature. This increase tends to a discontinuous jump in the limit $N \to \infty$.

Both the energy and magnetisation show an abrupt change close to the critical temperature, demonstrating the presence of a phase change. In each plot, however, the phase change appears as a continuous transition, rather than the discontinuity one would expect at the critical temperature. These are artefacts of the finite nature of the lattice being simulated. As expected, the magnetisation shows a sharp change at the critical temperature. Therefore, it was used to find an approximation of the critical temperature as $T_c = (2.35 \pm 0.10)$ J/k_B from taking T_c as the steepest part of the curve, and averaging over multiple samples.[2]

Subject to a first order Taylor series expansion, Onsager's analytical solution of the magnetism varies as $M \propto (T_c - T)^{\beta}$ as the temperature approaches T_c from below. Using a log-log fit in conjunction with

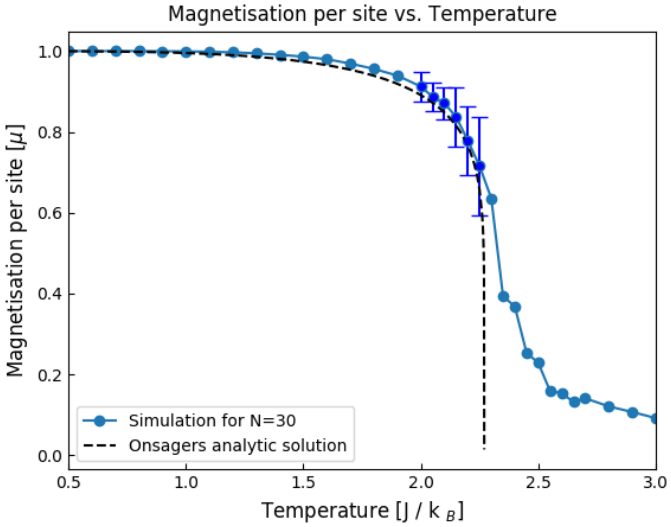

Figure 7: Magnetisation per site vs temperature for a 30x30 lattice, shows a significant change in magnetisation close to T_c. The form of the magnetisation tends to that of the analytical solution, again, in the limit $N \to \infty$.

linear regression via a curve fitting algorithm (and using the correct critical temperature for the lattice size used of N=30) an estimate of β was found as $\beta = 0.133 \pm 0.008$ [J^{-1}][3]. Only the magnetisations and their standard deviations in the temperature range [2, 2.25] were used to estimate β since the expression for magnetisation given above is only accurate for temperatures just below T_c. β is known as a critical exponent and is predicted to be universal, meaning that all systems in the same universality class will show the same value of such a critical exponent. A universality class is defined by three factors: the range of the interactions (short for the Ising model), the dimensionality of the physical space (2D), and the number of components in the order parameter (one).

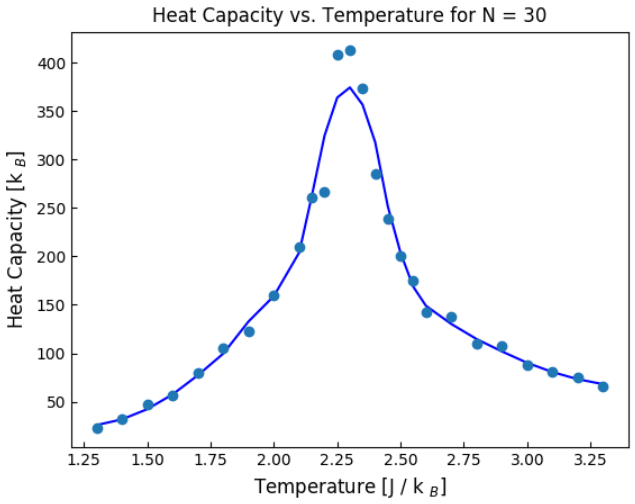

Figure 8: Heat capacity vs temperature for a 30x30 lattice, showing divergent behaviour at the critical temperature. A Savitzkay-Golay filter (blue curve) was applied to better show the divergent behaviour and for a better estimate of the critical temperature: $T_c = 2.3 \pm 0.05$.

[2] The error here is the absolute error, as opposed to the standard deviation due to the crude nature of the estimation.

[3] The curve fitting algorithm used throughout this paper is *scipy.curve_fit*, from the python library scipy.

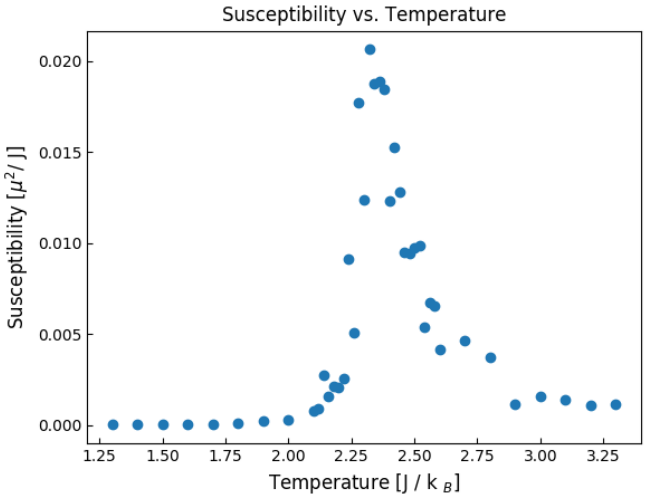

Figure 9: Magnetic susceptibility vs. temperature for a 30 x 30 lattice, demonstrating that fluctuations in the magnetisation also diverge at the critical temperature.

Table 1 summarises the estimates of T_c acquired from the divergence of the heat capacity for different lattice sizes.

Table 1: Estimates of the critical temperature along with its standard deviation for different lattice sizes (NxN).

N	T_c [J/k$_B$]	σ_{T_c}[J/k$_B$]
16	2.314	0.026
20	2.304	0.018
24	2.288	0.016
36	2.284	0.016
48	2.280	0.014
64	2.275	0.015

This data was then analysed using finite-size scaling, using a curve fitting algorithm to fit this data to the functional form $T_c(N) = T_c(\infty) + aN^{-1/\upsilon}$ such that an estimate of the critical temperature for an infinite lattice could be acquired.

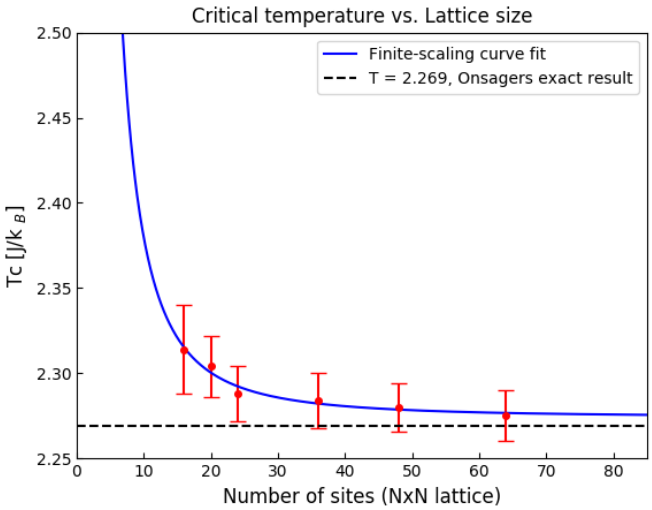

Figure 10: Critical temperature vs. lattice size with a finite-size scaling curve fit to find an estimate of the critical temperature for an infinitely large lattice.

T_c was found to decrease with N as $1/N^2$ where $\upsilon = 0.5 \pm 0.2$. The final estimate of T_c was found to be $T_c = 2.274 \pm 0.005$ [J/k$_B$] which agrees to within one standard deviation of Onsager's analytical result of $T_c = 2.269$.

4.2 NON-ZERO FIELD (H ≠ 0)

The application of a non-zero magnetic field resulted in a non-zero magnetisation pointing in the same direction as and increasing in strength with the field, even above the critical temperature. In the absence of a field and below the critical temperature there were two global minima, all spins pointing up or all spins pointing down, both of which were equally likely.

The introduction of the field shifted one of the global minima higher in energy and shifted the other lower in energy, such that the system was found to have one global minimum and one highly metastable local equilibrium coexisting with the ground state. Essentially, the field has broken the symmetry of the system under the 180° rotation of spins.

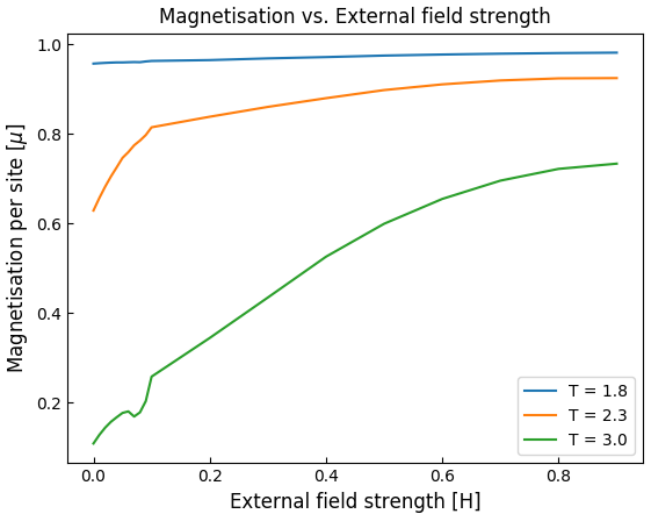

Figure 11: Absolute magnetisation per site vs. field strength at different temperatures for a 20x20 lattice, illustrating the increase of magnetisation with field strength.

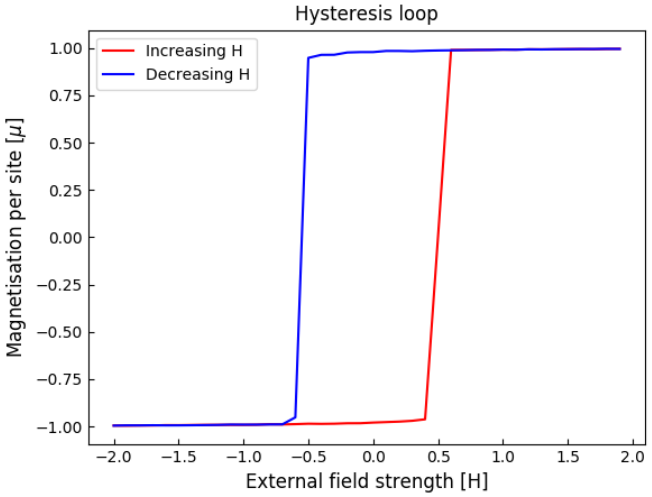

Figure 12: Magnetisation per site vs. field strength for a 16x16 lattice at $T<T_c$, forming a hysteresis loop. The hysteresis is due to a phase transition from the metastable state to the true equilibrium occurring at different values of the applied field.

5 CONCLUSION

The two dimensional Ising model was simulated using Monte Carlo techniques, namely, the Metropolis algorithm. A second order phase transition was observed at a critical temperature; below this temperature the spins were fully aligned in equilibrium and above it they were randomly oriented. Quantities such as magnetisation, energy, heat capacity and magnetic susceptibility were investigated around this critical temperature and from them estimations of the critical temperature were made for different lattice sizes. An estimate of the critical exponent β was derived from the magnetisation as $\beta = 0.133 \pm 0.008$ [J^{-1}], which agrees to within one standard deviation of the theoretical result for the Ising model, $\beta = 1/8$. The primary mechanism used to estimate the critical temperature was the heat capacity, owing to its divergence at the critical temperature. Analysis via finite-size scaling then led to the estimation of $T_c(\infty)$ (the critical temperature in the thermodynamic limit of an infinitely large lattice) as $T_c = 2.274 \pm 0.005$ [J/k_B], which agrees with Onsager's analytical prediction of $T_c = 2/(1 + \ln\sqrt{2})$ to within one standard deviation. Below the critical temperature, the application of an external magnetic field broke the symmetry of the system, such that the number of global minima was reduced from two to one. Above the critical temperature there was a non-zero magnetism in the direction of and increasing in strength with the field.

6 REFERENCES

[1] L. Onsager, Physical Review 65, 117 (1944).
[2] D. Landau, Physical Review B 13, 2997 (1976).
[3] K. Christensen and N. R. Moloney, Complexity and criticality (Imperial College Press, London, 2005).
[4] Ising Model, Rajesh Singh, accessed: 02/04/2018, URL https://rajeshrinet.github.io/blog/2014/ising-model/
[5] Paul Secular, Monte-Carlo simulation of small 2D Ising lattice with Metropolis dynamics (2015).

7 APPENDIX

7.1 APPENDIX A

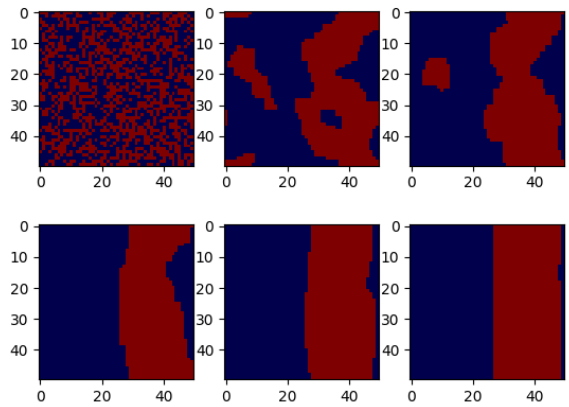

Figure 13: The spin orientations for a 50x50 lattice over 350 sweeps at T = 0.5 [J/k_B], J > 0. The spin configuration has settled into a local minimum. Another way to avoid this other than always starting in a uniform state would be to perform simulated annealing (slowly cooling the lattice to the desired final temperature).

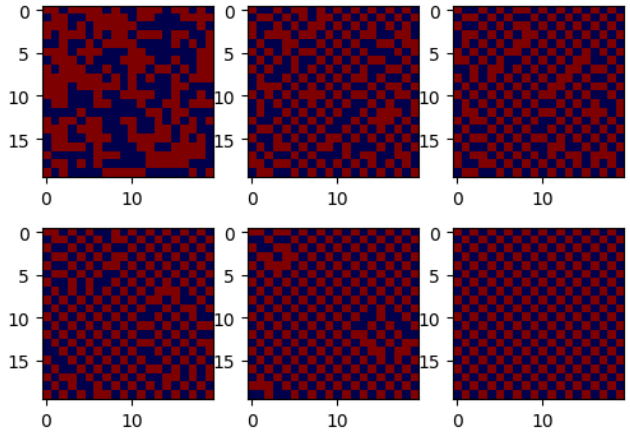

Figure 14: The spin orientations for a 20x20 lattice over 40 sweeps at T = 0.1 [J/k_B], J<0. The spin configuration has favoured anti-alignment, as expected (i.e. is anti-ferromagnetic).

WORD COUNT: 2580

7.2 APPENDIX B: THE PYTHON SOURCE CODE

```
"Ising model simulation of a two dimensional lattice of spins"

import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import scipy.signal
from numpy import exp, sinh, log, sqrt
from matplotlib import colors
from scipy.optimize import curve_fit
import time

"""
Create initial state of randomly orientated spins (+1 or -1)
"""
initialiseRandom = lambda N, T: 2 * np.random.randint(2, size=(N, N)) - 1

"""
Create initial state of up orientated spins (+1)
"""
initialiseOnes = lambda N, T: np.ones((N, N), dtype=int)

"""
Depending on temperature create initial state of up or randomly orientated spins
"""
initialiseByT = lambda N, T: initialiseRandom(N, T) if T >= 2.28 else initialiseOnes(N, T)


def spin_flip_energy(N, J, mu, H, state, i, j):
    """
        Compute energy to flip each state (using periodic boundary conditions)

    :param N: number sites, giving a lattice of N^2 spins
    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param state: configuration of the lattice of spins
    :param i: row
    :param j: column
    :return: energy required to flip the spin
    """
    #set boundary conditions

    left_state, right_state = state[i][(j -1) % N], state[i][(j + 1) % N]
    top_state, bottom_state = state[(i - 1) % N][j], state[(i + 1) % N][j]
    sum_Sj = left_state + right_state + top_state + bottom_state

    return 2 * J * state[i][j] * sum_Sj + mu * H * state[i][j]


def boltzmann_factor(delta_E, T, H, boltz4J, boltz8J):
    """
        Computes the boltzmann factor

    :param delta_E: energy required to flip the spin
    :param T: temperature
    :param H: magnetic field strength
    :param boltz4J: pre-set possible values of the boltzmann factor for optimisation
    :param boltz8J: pre-set possible values of the boltzmann factor for optimisation
    :return: the boltzmann factor for the given temperature and temperature and energy
    """
    boltzmann = 0
    if H == 0:
        if delta_E == 0:
            boltzmann = 1.0
        elif delta_E == 4:
            boltzmann = boltz4J
        elif delta_E == 8:
            boltzmann = boltz8J
    else:
        boltzmann = exp(-delta_E / T)
    return boltzmann
```

```
def state_change(N, state, T, J, mu, H):
    """
        Evolves the lattice in 'time' with one Monte Carlo sweep

    :param N: number sites, giving a lattice of N^2 spins
    :param state: configuration of the lattice of spins
    :param T: temperature
    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :return: updated state after one Monte Carlo sweep
    """

    # possible values of boltzmann factor: boltz4 corresponds to boltzmann factor for an
    # energy change of 4J, only possible positive energy changes are 0J, 4J and 8J.
    boltz4J = exp(- 4 / T)
    boltz8J = exp(- 8 / T)

    # create arrays of random numbers to randomly select sites as we step through the lattice.
    iRnd = np.random.randint(N, size=N ** 2)
    jRnd = np.random.randint(N, size=N ** 2)
    pRnd = np.random.uniform(0, 1, size=N ** 2)

    for k in range(N ** 2):
        i, j = iRnd[k], jRnd[k]
        delta_E = spin_flip_energy(N, J, mu, H, state, i, j)
        p = pRnd[k]
        boltzmann = boltzmann_factor(delta_E, T, H, boltz4J, boltz8J)
        if delta_E < 0:
            state[i][j] *= -1
        elif boltzmann >= p:
            state[i][j] *= -1
    return state


def energy(N, state, J, mu, H):
    """
        Computes the total energy of the lattice

    :param N: number sites, giving a lattice of N^2 spins
    :param state: configuration of the lattice of spins
    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :return: total energy of the lattice
    """
    energy = 0
    for i in np.arange(0, N):
        for j in np.arange(0, N):
            left_state, right_state = state[i][(j - 1) % N], state[i][(j + 1) % N]
            top_state, bottom_state = state[(i - 1) % N][j], state[(i + 1) % N][j]
            sum_Sj = left_state + right_state + top_state + bottom_state
            E_site = -J * state[i][j] * sum_Sj - mu * H * state[i][j]
            energy += E_site
    return energy / 4          #factor of 1/4 accounts for over-counting counting of interaction energies


def find_magnetisation(initialise, T, N, num_sweeps, J, mu, H, equilibrate):
    """
        Equilibrates the lattice then computes the average magnetisation per site

    :param initialise: initialise the lattice
    :param T: temperature
    :param N: number sites, giving a lattice of N^2 spins
    :param num_sweeps: number of sweeps to average over
    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param equilibrate: number of sweeps to equilibrate the lattice
    :return: magnitude of the average magnetisation per site, an array of the samples
             of the absolute magnetisations
    """
    mag_vector = np.zeros(num_sweeps)
    state = initialise(N, T)

    for eq in range(equilibrate):
        state_change(N, state, T, J, mu, H)

    for sweep in range(num_sweeps):
```

```
        state = state_change(N, state, T, J, mu, H)
        mag_vector[sweep] = abs(np.mean(state))

    magnetisation = np.mean(mag_vector)

    return magnetisation, mag_vector


def autocorrelation(initialise, T, N, taulength, num_sweeps, J, mu, H, equilibrate):
    """
        Computes the autocorrelation and lag time

    :param initialise: initialise the lattice
    :param T: temperature
    :param N: number sites, giving a lattice of N^2 spins
    :param taulength: time scale to show magnetisation autocorrelation over
    :param num_sweeps: number of sweeps to average over
    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param equilibrate: number of sweeps to equilibrate the system
    :return: magnetisation autocorrelation vector, time lag - the number of sweeps for autocorrelation to
             decay to 1/e of the initial value
    """

    mean_mag = find_magnetisation(initialise, T, N, num_sweeps, J, mu, H, equilibrate)[0]
    mag_vector = np.array(find_magnetisation(initialise, T, N, num_sweeps, J, mu, H, equilibrate)[1])
    tau_range = np.array(range(taulength))
    A = np.zeros(taulength)
    tau_efold = []
    Mtau = mag_vector[taulength:]
    Mtau_prime = Mtau - mean_mag

    A[0] = abs(np.mean(np.square(Mtau_prime)))

    for tau in range(1, taulength):

        #M is magnetisation snapshot array truncated by tau elements, Mtau is the snapshot array without the
        #first tau elements, and the primes correspond to taking the fluctuations of these about the mean
        Mprime = mag_vector[taulength - tau_range[tau]: - tau_range[tau]] - mean_mag

        A[tau] = abs(np.mean(np.multiply(Mprime, Mtau_prime)))
        tau_efold.append(A[tau])

    #now normalise to get autocorrelation
    a = A / A[0]

    #etau (the time lag) indicates the time required for fluctuations of the lattice about the mean to
    #become negligible so gives an indication of the number of sweeps we would need to average over to get
    #accurate readings
    etau = curve_fit(lambda x, b: exp(- x / b), tau_range, a)[0]

    return a, etau


def hysteresis_eq(N, T, J, mu, equilibrate, num_sweeps, Hrange):
    """
        Provides the magnetisation of the system each time the state is equilibrated at a different H

    :param N: number sites, giving a lattice of N^2 spins
    :param T: temperature
    :param J: exchange energy
    :param mu: magnetic permeability
    :param equilibrate: number of sweeps to equilibrate the system
    :param num_sweeps: number of sweeps to average over
    :param Hrange: dictates whether H goes from negative to positive, or positive to negative
    """
    state = initialiseRandom(N, T)
    mag = np.zeros(len(Hrange))
    for i in range(len(Hrange)):
        H = Hrange[i]
        for eq in range(equilibrate):
            state_change(N, state, T, J, mu, H)

        m = np.zeros(num_sweeps)
        for sweep in range(num_sweeps):
            state_change(N, state, T, J, mu, H)
            m[sweep] = np.mean(state)
        mag[i] = np.mean(m)

    return mag
```

```
def hysteresis(J, mu, num_sweeps=80, N=16, T=1.6, equilibrate=100):
    """
        Shows how the magnetisation changes as the field is slowly vaired (i.e. allowing the system to
        equilibrate each time the field is changed)

    :param J: exchange energy
    :param mu: magnetic permeability
    :param num_sweeps: number of sweeps to average over
    :param N: number sites, giving a lattice of N^2 spins
    :param T: temperature
    :param equilibrate: number of sweeps to equilibrate the system
    """
    H_increase = np.arange(-2, 2, 0.1)

    #Reverse the array such that the field decreases from a positive value to a negative one
    H_decrease = H_increase[::-1]

    mag = hysteresis_eq(N, T, J, mu, equilibrate, num_sweeps, H_increase)
    plt.plot(H_increase, mag, 'r', label='Increasing H')

    mag = hysteresis_eq(N, T, J, mu, equilibrate, num_sweeps, H_decrease)
    plt.plot(H_decrease, mag, 'b', label='Decreasing H')

    plt.legend(loc='best')
    plt.xlabel('External field strength [H]', fontsize='12')
    plt.ylabel('Magnetisation per site [' + r'$\mu$]', fontsize='12')
    plt.tick_params(direction='in', which='major')
    plt.title('Hysteresis loop')


def external_field_mag(J, mu, N=20):
    """
        Shows how the absolute magnetisation per site varies with external field strength

    :param J: exchange energy
    :param mu: magnetic permeability
    :param N: number sites, giving a lattice of N^2 spins
    """
    H_field = np.append(np.arange(0, 0.1, 0.01), np.arange(0.1, 1, 0.1))
    T_range = np.array([1.8, 2.3, 3])
    H_length = len(H_field)
    T_length = len(T_range)

    for i in range(T_length):
        print(str(i ) + '/' +str(len(T_range)))
        mag = np.zeros(H_length)
        temp = T_range[i]
        if temp > 2.2 and temp < 2.6:
            num_sweeps = 4000
            equilibrate = N ** 2
        else:
            num_sweeps = 400
            equilibrate = int(0.25 * N ** 2)

        for j in range(H_length):
            H = H_field[j]
            mag[j] = find_magnetisation(initialiseByT, temp, N, num_sweeps, J, mu, H, equilibrate)[0]
        H_filter = scipy.signal.savgol_filter(mag, 9, 2)

        plt.plot(H_field, H_filter, label= 'T = ' +str(temp))
    plt.legend(loc='best')
    plt.xlabel('External field strength [H]', fontsize='12')
    plt.ylabel('Magnetisation per site [' + r'$\mu$]', fontsize='12')
    plt.tick_params(direction='in', which='major')
    plt.title('Magnetisation vs. External field strength')


def finite_scaling():
    """
        Shows how the critical temperature varies with lattice size, and provides and estimate of the true
Tc
        (i.e. in the limit of an infinitely large lattice).

    """
    #Critical temperatures
    N      = np.array([ 16,    20,    24,    36,    48,   64])
    T_mean = np.array([2.320, 2.312, 2.300, 2.292, 2.289, 2.283])
    T_err  = np.array([0.013, 0.012, 0.016, 0.016, 0.014, 0.012])
```

```
    popt, var = curve_fit(lambda x, Tc_inf, a, nu: Tc_inf + a* x ** (-1 / nu), N, T_mean, sigma=T_err)
    Tc_inf, a, nu = popt
    Tc_err, a_err, nu_err = np.sqrt(np.diag(var))

    N_range = np.arange(1, 85, 0.1)
    T_fit = Tc_inf + a * np.power(N_range, (-1 / nu))

    plt.errorbar(N, T_mean, yerr=T_err, fmt='o', color='r', markersize=4, capsize=5, ecolor='r')
    plt.plot(N_range, T_fit, 'b', label='Finite-scaling curve fit')
    plt.axhline(y=2.269, color='k', linestyle='--', label='T = 2.269, Onsagers exact result')
    plt.legend(loc='best')
    plt.ylim(2.25, 2.5)
    plt.xlim(0, 85)
    plt.xlabel('Number of sites (NxN lattice)', fontsize='12')
    plt.ylabel('Tc [J/k' + r'$\ _B$]', fontsize='12')
    plt.tick_params(direction='in', which='major')
    plt.title('Critical temperature vs. Lattice size')
    print('Tc = ' + str(Tc_inf) + ' +/- ' + str(Tc_err))
    print('nu = ' + str(nu) + ' +/- ' + str(nu_err))


def autocorrelation3(J, mu, H, taulength=50):
    """
        Shows how the time lag varies for different temperatures and lattice sizes together

    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param taulength: time scale to show magnetisation autocorrelation over
    """
    T_range = np.append(np.append(np.arange(2.0, 2.1, 0.1), np.arange(2.15, 2.4, 0.05)),
                        np.arange(2.4, 3, 0.1))
    lag_time = np.zeros(len(T_range))
    N_range = np.array([16, 24])

    for N in N_range:
        for i in range(len(T_range)):
            temp = T_range[i]
            if temp > 2.1 and temp < 2.5:
                num_sweeps = 50000
                equilibrate = 4 * N ** 2
            else:
                num_sweeps = 10000
                equilibrate = int(N ** 2)
            print(str(i) + '/' + str(len(T_range)))
            lag_time[i] = autocorrelation(initialiseByT, temp, N, taulength, num_sweeps, J, mu, H,
                                          equilibrate)[1]
        plt.plot(T_range, lag_time, 'o', linestyle='-', label='N=' + str(N))
    plt.xlabel('Temperature [J / k' + r'$\ _B$]', fontsize='12')
    plt.ylabel('Time lag (equilibration time)', fontsize='12')
    plt.legend(loc='best')
    plt.tick_params(direction='in', which='major')
    plt.title('Autocorrelation time lag vs. Temperature')


def autocorrelation2(J, mu, H, num_sweeps=2000, T=2, equilibrate=600, taulength=40):
    """
        Shows exponential decay of autocorrelation for a set temperature and different N's

    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param num_sweeps: number of sweeps to average over
    :param T: temperature
    :param equilibrate: number of sweeps to equilibrate the system
    :param taulength: time scale to show magnetisation autocorrelation over
    """
    tau_range = np.array(range(taulength))
    N_range = np.array([16, 24])

    for N in N_range:
        a = autocorrelation(initialiseByT, T, N, taulength, num_sweeps, J, mu, H, equilibrate)[0]
        plt.plot(tau_range, a, label='N=' + str(N))

    plt.legend(loc='best')
    plt.xlabel('Time (tau)', fontsize='12')
    plt.ylabel('Autocorrelation', fontsize='12')
    plt.tick_params(direction='in', which='major')


def autocorrelation1(J, mu, H, num_sweeps=2000, N=16, equilibrate=1000, taulength=50):
```

```
    """
        Shows exponential decay of autocorrelation for a set N and different temperatures

    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param num_sweeps: number of sweeps to average over
    :param N: number sites, giving a lattice of N^2 spins
    :param equilibrate: number of sweeps to equilibrate the system
    :param taulength: time scale to show magnetisation autocorrelation over
    """
    tau_range = np.array(range(taulength))
    T_range = np.array([2, 2.3, 2.7])

    for i in range(len(T_range)):
        print(str(i) + '/' + str(len(T_range)))
        temp = T_range[i]
        a = autocorrelation(initialiseByT, temp, N, taulength, num_sweeps, J, mu, H, equilibrate)[0]
        plt.plot(tau_range, a, label='T=' + str(temp))

    plt.legend(loc='best')
    plt.xlabel('Time (tau)', fontsize='12')
    plt.ylabel('Autocorrelation', fontsize='12')
    plt.tick_params(direction='in', which='major')
    plt.title('Autocorrelation vs. Time')


def susceptibility_plot(J, mu, H, N=20):
    """
        Shows how magnetic susceptibility varies with temperature and provides an estimate of Tc

    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param N: number sites, giving a lattice of N^2 spins
    """
    T_range = np.append(np.append(np.arange(1.3, 2.1, 0.1), np.arange(2.1, 2.6, 0.05)),
                        np.arange(2.6, 3.4, 0.1))
    chi = np.zeros(len(T_range))

    for i in range(len(T_range)):
        temp = T_range[i]
        print(str(i) + '/' + str(len(T_range)))

        if temp > 2.1 and temp < 2.6:
            num_sweeps = 4000
            equilibrate = N ** 2
        else:
            num_sweeps = 700
            equilibrate = int(0.25 * N ** 2)

        magnetisation, mag_vector = find_magnetisation(initialiseByT, temp, N, num_sweeps, J, mu, H,
                                                       equilibrate)
        chi[i] = (np.std(mag_vector) ** 2) / temp

        if chi[i] == max(chi):
            imax = i

    Tc_estimate_SC = T_range[imax]
    print('Curie temperature estimate = ', Tc_estimate_SC)

    plt.xlabel('Temperature [J / k' + r'$\ _B$]', fontsize='12')
    plt.ylabel('Susceptibility ['r'$\mu^2$' + '/ J]', fontsize='12')
    plt.title('Susceptibility vs. Temperature')
    plt.tick_params(direction='in', which='major')
    plt.plot(T_range, chi, 'o')


def heat_capacity(J, mu, H, N=30):
    """
        Shows how heat capacity varies with tmeperature and provides and estimate of Tc

    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param N: number sites, giving a lattice of N^2 spins
    """
    T_range = np.append(np.append(np.arange(1.3, 2.1, 0.1), np.arange(2.1, 2.5, 0.02)),
                        np.arange(2.5, 3.4, 0.1))
    spec_heat = np.zeros(len(T_range))
```

```
    for i in range(len(T_range)):
        print(str(i) + '/' + str(len(T_range)))
        temp = T_range[i]
        state = initialiseByT(N, temp)

        if temp > 2.1 and temp < 2.6:
            num_sweeps = 4000
            equilibrate = N ** 2
        else:
            num_sweeps = 600
            equilibrate = int(0.25 * N ** 2)

        energy_samples = np.zeros(num_sweeps)
        for eq in range(equilibrate):
            state_change(N, state, temp, J, mu, H)

        for sweep in range(num_sweeps):
            state_change(N, state, temp, J, mu, H)
            energy_samples[sweep] = energy(N, state, J, mu, H)

        spec_heat[i] = ((np.std(energy_samples)) / temp) ** 2

    C_filter = scipy.signal.savgol_filter(spec_heat, 9, 2)

    # Find T value corresponding to the maximum in the filtered heat capacity curve
    Tc = T_range[C_filter.argmax()]

    print('Tc for N = ' + str(N) + ' is', Tc)
    plt.plot(T_range, C_filter, 'b')
    plt.plot(T_range, spec_heat, 'o')
    plt.xlabel('Temperature [J / k' + r'$\ _B$]', fontsize='12')
    plt.ylabel('Heat Capacity [k' + r'$\ _B$]', fontsize='12')
    plt.title('Heat Capacity vs. Temperature for N = ' + str(N))
    plt.tick_params(direction='in', which='major')


def energy_plot(J, mu, H, N=36):
    """
        Shows how energy per site varies with temperature

    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    """
    T_range = np.append(np.append(np.arange(1.2, 2.1, 0.1), np.arange(2.1, 2.6, 0.05)), np.arange(2.6, 4,
0.1))

    Energy0 = []

    for i in range(len(T_range)):
        temp = T_range[i]
        state = initialiseByT(N, temp)
        E1 = 0
        print(str(i) + '/' + str(len(T_range)))

        if temp > 2.2 and temp < 2.6:
            num_sweeps = 3200
            equilibrate = N ** 2
        else:
            num_sweeps = 800
            equilibrate = int(0.025 * N ** 2)

        scale = 1 / (N * N * num_sweeps)

        for eq in range(equilibrate):
            state_change(N, state, temp, J, mu, H)

        for sweep in range(num_sweeps):
            state_change(N, state, temp, J, mu, H)
            Ene = energy(N, state, J, mu, H)
            E1 = E1 + Ene
        Energy0.append(E1 * scale)

    plt.plot(T_range, Energy0, 'o', linestyle='-', label='N=' + str(N))
    plt.tick_params(direction='in', which='major')
    plt.xlabel('Temperature [J / k' + r'$\ _B$]', fontsize='12')
    plt.ylabel('Energy [J]', fontsize='12')
    plt.title('Energy per site vs. Temperature')
    plt.legend(loc='best')
```

```
def mag_plot(J, mu, H, N=30):
    """
        Shows how magnetisation per site varies with temperature, and provides an estimate of Tc and beta

    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param N: number sites, giving a lattice of N^2 spins
    """
    T_range = np.append(np.append(np.arange(0.5, 2, 0.1), np.arange(2, 2.7, 0.05)),
                        np.arange(2.8, 3.1, 0.1))

    mag = np.zeros(len(T_range))
    mag_err = np.zeros(len(T_range))
    log_err = np.zeros(len(T_range))
    mag_array = []
    for i in range(len(T_range)):
        temp = T_range[i]
        print(str(i) + '/' + str(len(T_range)))

        if temp == 2.25:
            num_sweeps = 10000
            equilibrate = N ** 2
        elif temp > 1.9 and temp < 2.7:
            num_sweeps = 5000
            equilibrate = N ** 2
        else:
            num_sweeps = 400
            equilibrate = int(0.25 * N ** 2)

        mag[i] = find_magnetisation(initialiseByT, temp, N, num_sweeps, J, mu, H, equilibrate)[0]
        mag_vector = find_magnetisation(initialiseByT, temp, N, num_sweeps, J, mu, H, equilibrate)[1]
        mag_err[i] = np.std(mag_vector)
        log_err[i] = mag_err[i] / mag[i]

        if i > 0:
            mag_diff = mag[i] - mag[i - 1]
            mag_array.append(mag_diff)
            steepest_slope = min(mag_array)
            if mag_diff == steepest_slope:
                imax = i

    T_onsager = 2.27
    T_range2 = np.append(np.arange(0.5, 2.26, 0.01), np.arange(2.26, 2.27, 0.001))
    M_analytic = (1 - (sinh(log(1 + sqrt(2))) * T_onsager / T_range2) ** -4) ** (1 / 8)

    coeffs, var = curve_fit(lambda x, beta, const: const + beta * x, log(2.3 - T_range[15:-13]),
                            log(mag[15:-13]), sigma=log_err[15:-13])
    beta = coeffs[0]
    beta_err = np.sqrt(np.diag(var))[0]

    print('Beta = ' + str(beta) + ' +/- ' + str(beta_err))
    Tc_estimate = T_range[imax]
    print('Critical temperature for N = ' + str(N) + ' from steepest slope is', Tc_estimate)

    plt.xlabel('Temperature [J / k' + r'$\ _B$]', fontsize='12')
    plt.ylabel('Magnetisation per site [' + r'$\mu$]', fontsize='12')
    plt.tick_params(direction='in', which='major')
    plt.plot(T_range, mag, 'o', linestyle='-', label='Simulation for N=' + str(N))
    plt.errorbar(T_range[15:-13], mag[15:-13], yerr=mag_err[15:-13], fmt='o', color='b', markersize=4,
                 capsize=5, ecolor='b')
    plt.plot(T_range2, M_analytic, color='black', linestyle='--', label='Onsagers analytic solution')
    plt.xlim(0.5, 3)
    plt.title('Magnetisation per site vs. Temperature')
    plt.legend(loc='best')


def mag_equilibrium(J, mu, H, num_sweeps=300, T=1.2, equilibrate=0):
    """
        Shows how the magnetisation varies as the system is brought to equilibrium

    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param num_sweeps: number of sweeps to average over
    :param T: temperature
    :param equilibrate: number of sweeps to equilibrate the system
    """
    N_range = np.array([20, 40])
```

```
    for N in N_range:
        mag_vector = find_magnetisation(initialiseRandom, T, N, num_sweeps, J, mu, H, equilibrate)[1]
        plt.plot(range(num_sweeps), mag_vector, label='N = ' + str(N))

    plt.xlabel('Time [number of sweeps]', fontsize='12')
    plt.ylabel('Magnetisation per site [' + r'$\mu$]', fontsize='12')
    plt.tick_params(direction='in', which='major')
    plt.title('Magnetisation per site vs. number of sweeps, T = ' + str(T))
    plt.legend(loc='best')


def plot_spins(J, mu, H, N=30, T=0.1, equilibrate=200):
    """
        Shows how the spins evolve

    :param J: exchange energy
    :param mu: magnetic permeability
    :param H: magnetic field strength
    :param N: number sites, giving a lattice of N^2 spins
    :param T: temperature
    :param equilibrate: number of sweeps to equilibrate the system
    """
    state = initialiseRandom(N, T)
    cmap = matplotlib.cm.seismic
    bounds = [-1, 0, 1]
    norm = colors.BoundaryNorm(bounds, cmap.N)
    plt.subplot(231)
    plt.imshow(state, cmap=cmap, norm=norm)

    for sweep in range(equilibrate):
        state = state_change(N, state, T, J, mu, H)

        if sweep == int(0.05 * equilibrate):
            plt.subplot(232)
            plt.imshow(state, cmap=cmap, norm=norm)
        if sweep == int(0.1 * equilibrate):
            plt.subplot(233)
            plt.imshow(state, cmap=cmap, norm=norm)
        if sweep == int(0.2 * equilibrate):
            plt.subplot(234)
            plt.imshow(state, cmap=cmap, norm=norm)
        if sweep == int(0.5 * equilibrate):
            plt.subplot(235)
            plt.imshow(state, cmap=cmap, norm=norm)
        if sweep == int(equilibrate - 1):
            plt.subplot(236)
            plt.imshow(state, cmap=cmap, norm=norm)


def simulate(simulation, J=1, H=0, mu=1):
    start = time.time()
    if simulation == "spin_plot":
        plot_spins(J, mu, H)

    if simulation == "mag_equilibrium":
        mag_equilibrium(J, mu, H)

    if simulation == "mag_plot":
        mag_plot(J, mu, H)

    if simulation == "energy_plot":
        energy_plot(J, mu, H)

    if simulation == "heat_capacity":
        heat_capacity(J, mu, H)

    if simulation == "susceptibility_plot":
        susceptibility_plot(J, mu, H)

    if simulation == "autocorrelation1":
        autocorrelation1(J, mu, H)

    if simulation == "autocorrelation2":
        autocorrelation2(J, mu, H)

    if simulation == "autocorrelation3":
        autocorrelation3(J, mu, H)

    if simulation == "finite_scaling":
        finite_scaling()
```

```
    if simulation == "external_field_mag":
        external_field_mag(J, mu)

    if simulation == "hysteresis":
        hysteresis(J, mu)

    end = time.time()
    print("Program run time =", end - start)

    plt.show()


#simulate("spin_plot")
#simulate("mag_equilibrium")
#simulate("mag_plot")
#simulate("energy_plot")
#simulate("heat_capacity")
#simulate("susceptibility_plot")
#simulate("autocorrelation1")
#simulate("autocorrelation2")
#simulate("autocorrelation3")
#simulate("finite_scaling")
#simulate("external_field_mag")
#simulate("hysteresis")
```