Ising Model Investigations Using 2D and 3D Lattices With Long-range Interactions

2 April 2020

Abstract

A two-dimensional square lattice was used extensively to investigate several properties of ferromagnetic materials using the Ising model. The critical temperature T_c was calculated using finite size scaling and a Gaussian fit on the heat capacity as a function of temperature. It was found in both cases to converge to Onsager's analytical result ($\approx 2.26\ J/k_B$) with increasing lattice dimensions. The investigation also includes the hysteresis effect and the existence of meta-stable states, the behaviour of susceptibility with temperature and the formation of large grains near T_c. The susceptibility and heat capacity were found to take their maximum value at T_c and decay to zero away from it. It was also found that decorrelation time grows with lattice dimensions and at T_c it takes its maximum value. Long-range interactions in 2D gave $T_c \approx 5.5\ J/k_B$ and the 3D lattice phase transition was found to occur at $T_c \approx 4.5\ J/k_B$. Using supervised learning, a neural network was trained to predict the temperature of a given lattice configuration.

1 Introduction

The Ising model uses statistical physics to investigate properties of ferromagnetic materials. The material is modeled as a lattice of sites and each site holds a spin of either $+1$ or -1. In this investigation, the Monte Carlo Metropolis algorithm was used to evolve the system and collect data, utilizing Markov Chains and importance sampling. The use of Markov chains implies that the next sample collected each time only depends on the current state of the system. Importance sampling implies that the algorithm is based on the principle of detailed balance so that $p_A T_{AB} = p_B T_{BA}$ where p is the probability of being in a state and T_{AB} is the rate of transition between states A and B.

After its invention in 1920 by Wilhelm Lenz, the model has greatly evolved along with the creation of theoretical predictions that go with it [1]. Now, state-of-the-art AI algorithms gather data from the Ising model and are able to predict the phase of a sample of material with high accuracy [2].

The following section includes analysis and implementation of various subjects investigated. Section 3 contains theory and methods for each subject along

with its results presented in each subsection. Section 4 includes a brief discussion for the various subjects with some suggestions for improvement and section 5 is a summary of the investigation. Code listings can be found at the end of this paper.

2 Analysis & Implementation

Metropolis is the algorithm that drives all of the programs for this paper and is thoroughly described in [3]. For an N sites lattice, we choose N random sites and for each one we calculate if the spin is to be flipped. Since this requires the neighbours for its site at every iteration, access to neighbours should be $\mathcal{O}(1)$. Hence it was considered good practice to create a dictionary with keys as the lattice sites indices and values as lists of their neighbours' indices at the start of every program. To investigate long-range interactions we just include another dictionary for the next-nearest neighbours.

To avoid unnecessary calculations for every Metropolis sweep and increase efficiency, the difference in energy that would cause a spin flip was calculated once at the beginning of the program. The lattice is also generated once at the beginning and then simply changes state. Both 2D and 3D lattices are represented in Python by a one dimensional list ranging from 0 to N-1. For each sweep, Python's random package was used for the Metropolis function to generate the N random sites and get a number from 0 to 1 using a uniform distribution utilizing the built-in functions 'randint' and 'uniform'.

For computing the cluster sizes, breadth-first search was used which has $\mathcal{O}(N + E)$ time complexity, where E is the number of edges connecting neighbours with identical spin. Hence this algorithm's CPU time was dominated by the lattice size. For small N it was relatively fast, compared to the bootstrap method.

The bootstrap method described in section 3.2.1 provides a way to calculate statistical error in observables such as magnetisation or energy and carries a high time complexity. It contains two nested for-loops giving a time complexity $\mathcal{O}(n_{samples} \times n_{Bins})$ and is called at every temperature we want to collect data for the observables. Hence the heat capacity and susceptibility were the most expensive to calculate and form a plot against temperature.

A shallow neural network was built using the Keras library [4] with a simple network architecture described in 3.9.1. A more systematic approach to the architecture would have been Bayesian hyper-parameter optimisation (BHO), however the performance gave satisfactory results, hence BHO was not used. The multi-layered perceptron (MLP) was relatively fast to train due to the low complexity of the architecture - following the logic that the fewer assumptions a model makes, the better it can generalise to unseen data.

3 Theory, Methods & Results

3.1 Total magnetisation fluctuations and autocorrelation

Code used: avg_mag.py, ising_ca_reader.py, avg_mag_table.py, correlation.py, correlation_table.py

3.1.1 Theory & Methods

The total magnetisation M is given by [3]

$$M = \sum_{i=0}^{N-1} s_i \tag{1}$$

When the system is below the critical temperature T_c with a hot start, it will take some Monte Carlo sweeps (MCS) to thermalise to its average total magnetisation. After that, the system fluctuates about the mean and we quantify the fluctuations by taking the standard deviation, omitting samples before thermalisation.

The autocovariance and autocorrelation are respectively calculated for a time lag τ by [3]

$$A(\tau) = \langle (M(t) - \langle M(t) \rangle) \times (M(t+\tau) - \langle M(t+\tau) \rangle) \rangle \tag{2}$$

$$\alpha(\tau) = \frac{A(\tau)}{A(0)} \tag{3}$$

To determine the time lag τ_e over which the autcorrelation falls to $\frac{1}{e}$ we generate a list of τ values and their corresponding $\alpha(\tau)$. Since this is a discrete process we calculate the value of $\alpha(\tau)$ at $1/e$ of its initial value and find the τ which corresponds to the nearest value. τ_e was found for different values of temperature and N, the total number of lattice sites.

We also investigate critical slowing down which occurs near T_c using

$$\tau_e \sim L^z \tag{4}$$

where L is the dimension of the lattice and z is a critical exponent. Near the phase transition the correlation length ξ becomes large and since Metropolis has a localised effect, the system will get stuck in similar configurations with large clusters for more and more sweeps as L is increased [3].

3.1.2 Results

An example of total and average energy along with average magnetisation vs sweeps are presented below, which clearly depict the thermalisation process.

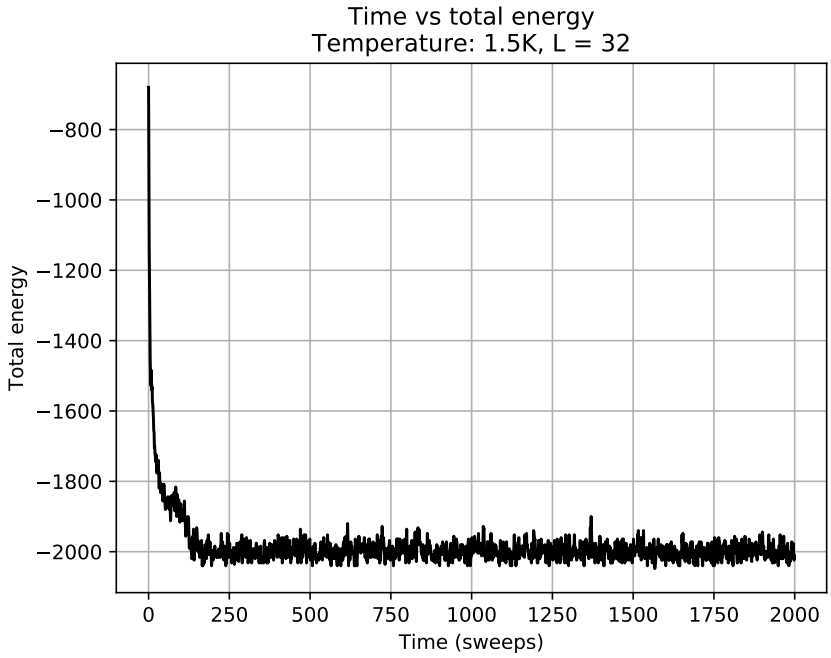


Figure 1: With a hot start the system takes around 250 sweeps at T = 1.5 J/k_B to thermalise.

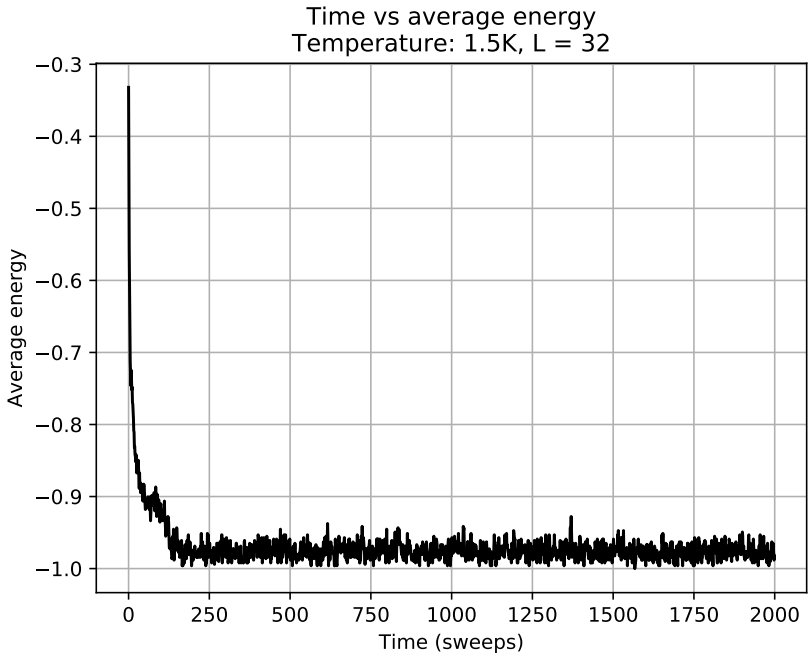


Figure 2: The maximum magnitude of energy per link is 1 and thermalisation time is the same as for the total energy.

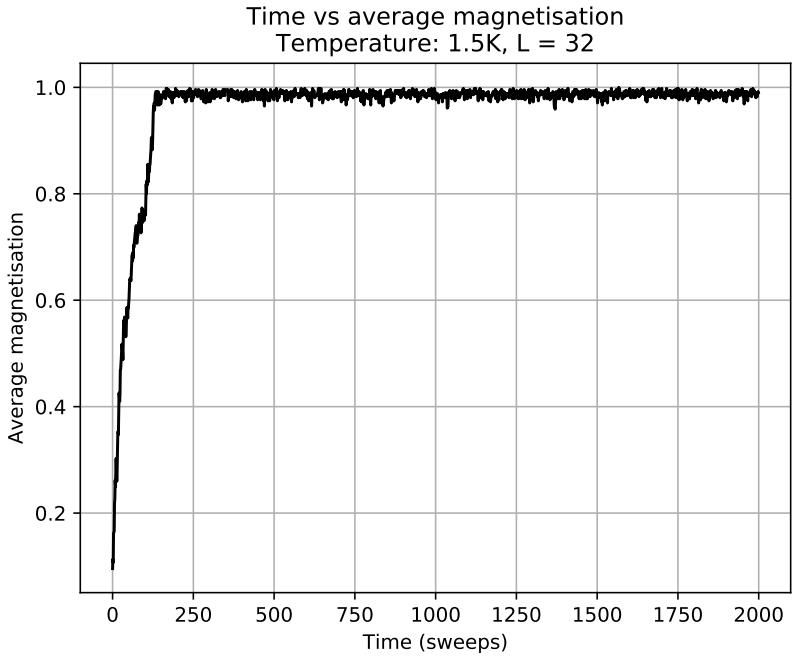


Figure 3: The average magnetisation is the magnetisation per site and its maximum magnitude is 1.

As we approach T_c, the total magnetisation fluctuations σ_M become bigger and for $T \to 0\ J/k_B$, $\sigma_M \to 0$. The system in Figure 4 which shows total magnetisation vs time has $L = 10$ with a hot start.

The process of finding τ_e is shown graphically in Figure 5 and Table 1 gives τ_e for different N and T.

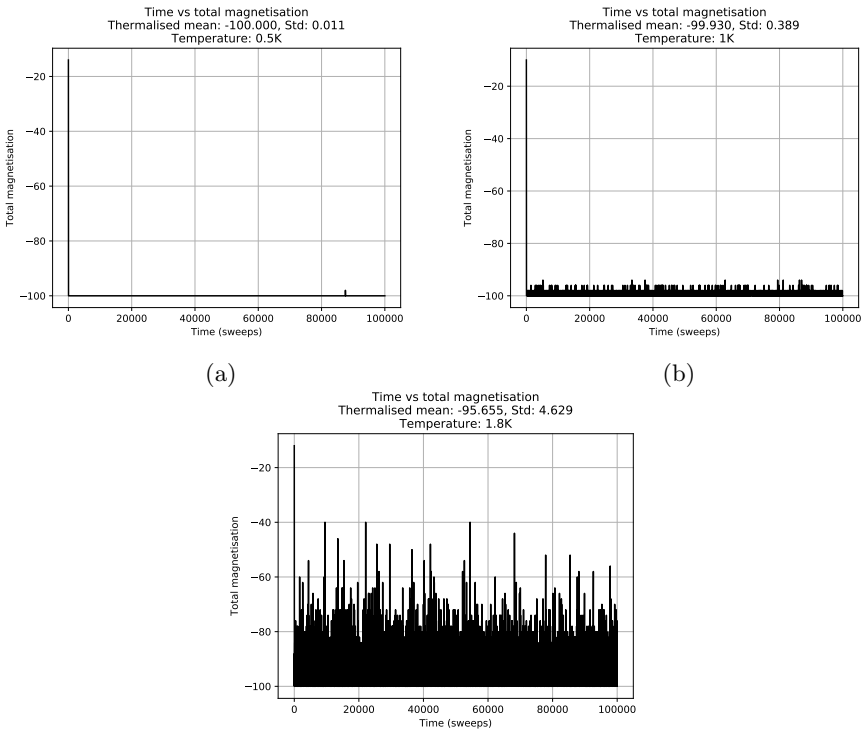


Figure 4: Total magnetisation vs time for $L = 10$ and a hot start. (a): T = 0.5 J/k_B, (b): 1 J/k_B, (c): 1.8 J/k_B with mean -100.000, -99.930, -95.655 and standard deviation 0.011, 0.389, 4.629 respectively.

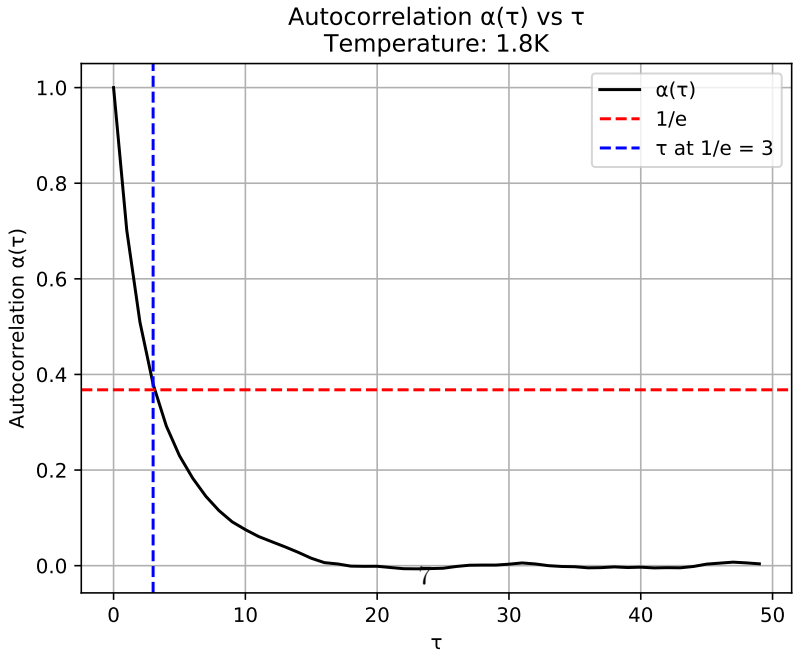


Figure 5: Finding τ_e which characterises the time scale for decorrelation. The plot shows the process for $T = 1.8$ J/k_B with $L = 10$ and a hot start.

L	**T** ($\mathbf{J}/k_B$)	τ_e
16	1.4	2
16	1.6	3
16	1.8	3
16	2.0	6
16	2.2	53
16	2.4	35
16	2.6	17
16	2.8	10
16	3.0	6
16	3.2	5
16	3.4	3
16	3.6	3
16	3.8	2
32	1.4	2
32	1.6	2
32	1.8	3
32	2.0	6
32	2.2	31
32	2.4	84
32	2.6	26
32	2.8	10
32	3.0	6
32	3.2	4
32	3.4	4
32	3.6	3
32	3.8	2
64	1.4	2
64	1.6	2
64	1.8	4
64	2.0	9
64	2.2	69
64	2.4	90
64	2.6	24
64	2.8	10
64	3.0	6
64	3.2	4
64	3.4	3
64	3.6	3
64	3.8	2

Table 1: Near T_c, τ_e becomes very large and increases with larger lattice dimensions due to critical slowing down. Away from T_c, the autocorrelation decreases fast and observables converge to a mean value much faster.

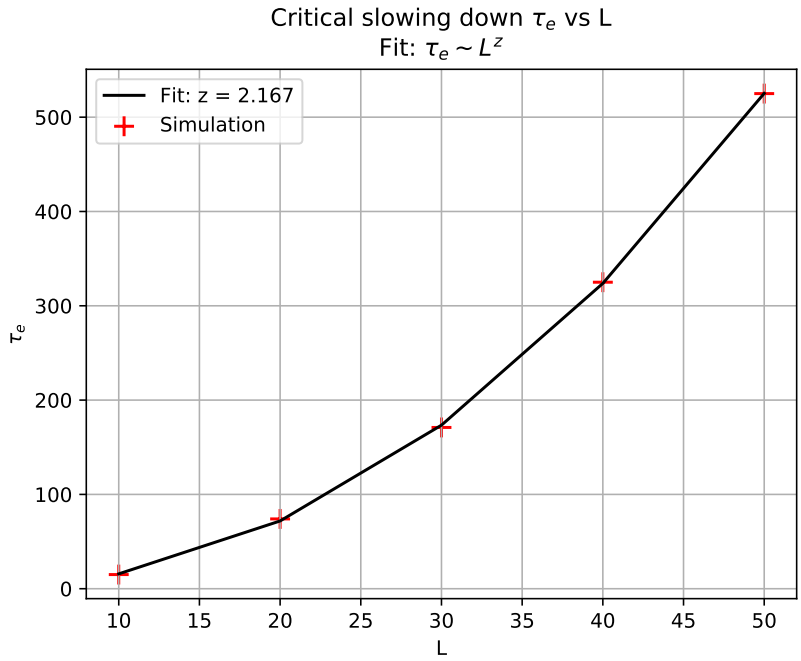
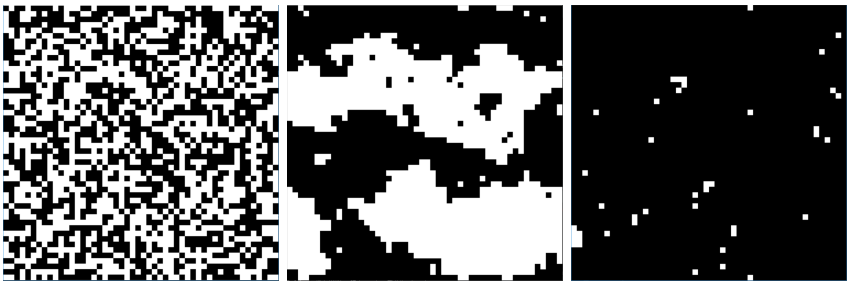


Figure 6: From the fitting of the points shown in this plot, the z exponent in equation (4) was found to be $z = 2.167 \pm 0.037$. The standard deviation is found from the covariance matrix of the fitting algorithm.



(a) (b) (c)

Figure 7: Evolution of cellular automata [5] under the Metropolis algorithm rules. This is the schematic representation of thermalisation at T = 1.7 with a hot start. (a) Hot start of randomly oriented spins, (b) After 100 Metropolis sweeps large clusters form, (c) Finally the state has thermalised to all spins pointing down with minor fluctuations. The animation can be viewed with ising_ca_reader.py.

3.2 Heat capacity and the critical temperature

Code used: bootstrap.py, heat_capacity.py, heat_capacity_plotter.py

3.2.1 Theory & Methods

The heat capacity is given by the fluctuation-dissipation theorem [6] as

$$C = \frac{\sigma_E^2}{k_B T^2} \tag{5}$$

where σ_E is the standard deviation of the energy calculated using the bootstrap method.

The bootstrap method [3] estimates the statistical error for an observable - in this case the energy. If we have collected n independent samples we create pseudo-data by randomly choosing n samples with replacement for each of the n_B bins we create. We then take the standard deviation for each bin and use that to calculate n_B values of heat capacity. The final heat capacity is the average and its standard deviation is derived from these n_B values using

$$\sigma_C = (1 + 2\tau_e)\sigma_{n_b} \tag{6}$$

where τ_e is the time constant for decorrelation.

The critical temperature is derived by fitting a Gaussian to the peak of the heat capacity plot - using scipy's curve_fit - and taking the mean of the Gaussian as T_c. The fit is weighted by the standard deviation of the points in the plot and provides a standard deviation for T_c, through the covariance matrix.

3.2.2 Results

The energy has a discontinuity in the first derivative with respect to temperature - since the model follows a second order phase transition - and hence in the heat capacity vs temperature plot we see that after T_c there is a discontinuous decrease, while before T_c the heat capacity increases faster than linearly as shown in Figure 8.

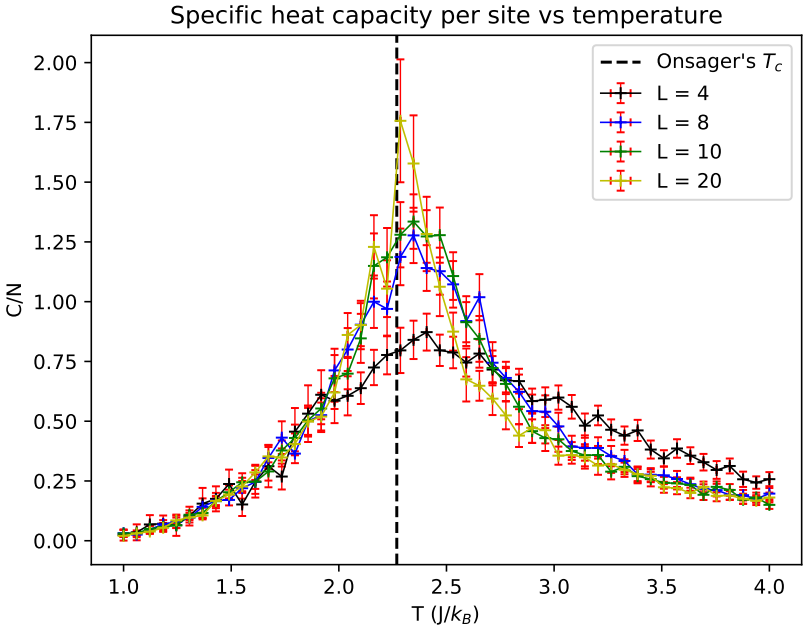


Figure 8: Heat capacity per site vs temperature for different lattice dimensions. As the dimension is increased the sharp change of behaviour near T_c becomes more clear. The error near T_c is always larger and the heat capacity tends to 0 as the temperature tends to 0 and ∞.

The data for the plot in Figure 9 were used to investigate finite size scaling in section 3.3 and generate values T_c for different lattice dimensions. Table 2 summarizes these values and shows how T_c converges to Onsager's analytical result as L is increased. The largest L used was 48 with $T_c = 2.27 J/k_B$ giving a 0.29 percentage difference from Onsager's result.

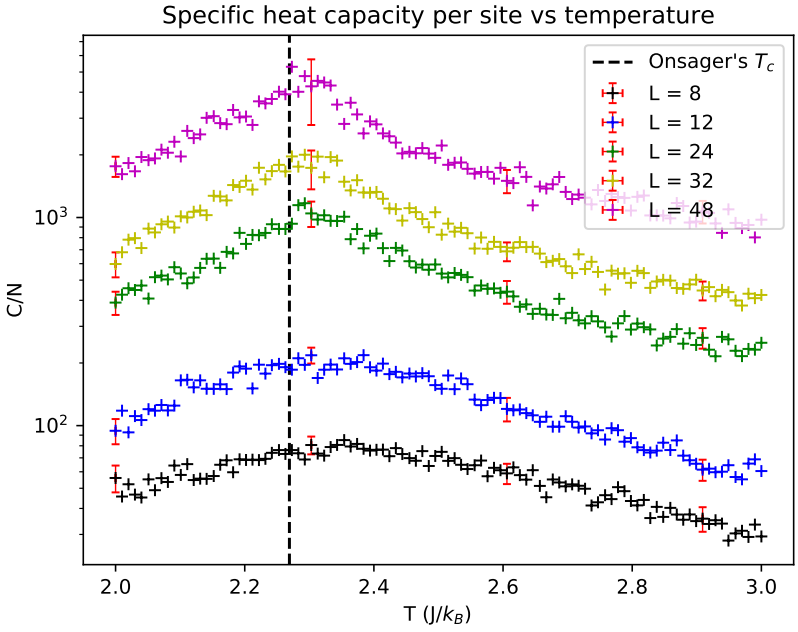


Figure 9: A log scale on the y-axis provides clarity for the behaviour near T_c as the lattice dimension is increased.

L	T_c	σ	**Percentage diff. %**
8	2.3513	0.0156	3.61
12	2.3246	0.0297	2.44
24	2.3013	0.0067	1.41
32	2.2877	0.0067	0.81
48	2.2758	0.0096	0.29

Table 2: L is the dimension of the lattice, T_c is the critical temperature, σ is the error in T_c and the last column gives the percentage difference between T_c and Onsager's analytical result for the critical temperature which is 2.2692 J/k_B.

3.3 Finite size scaling

Code used: bootstrap.py, heat_capacity.py, heat_capacity_plotter.py

3.3.1 Theory & Methods

Finite size scaling in the 2D Ising Model is the process of calculating critical exponents and the phase transition temperature using increasing values of L

- the lattice dimension [7]. The values of T_c for every L were calculated as described in 2.2.1 and presented in 2.2.2 with their associated errors. The equation used with the latter data is

$$T_c(L) = T_c(\infty) + \alpha L^{-\frac{1}{\nu}} \tag{7}$$

where α and ν are constant exponents and $T_c(\infty)$ is the critical temperature as the dimension of the system goes to infinity. To find the unknown values of equation (7), python's curve_fit function was used, which also provided the appropriate errors.

In Figure 11 the average magnetisation is plotted against temperature for different L values to investigate spontaneous magnetisation at T_c where the phase transition occurs and also observe how L affects this behaviour.

3.3.2 Results

$T_c(\infty)$ was found to be $2.264 \pm 0.152 J/k_B$ and $\nu = 1.12 \pm 0.15$. It is apparent that with more data at larger N the fit would have been able to capture the horizontal asymptote giving a more accurate $T_c(\infty)$. Onsager's analytical result for the critical temperature is $T_c = \frac{2}{ln(1+\sqrt{2})} = 2.2692$. The result from the finite size scaling has a 0.22 percentage difference which is smaller than the Gaussian fit method used in section 3.2.

The spontaneous magnetisation in Figure 11 is seen to occur near Onsager's predicted T_c, however how quickly the phase transition occurs is determined by the dimension of the lattice. The higher the L, the sharper the transition - which defines T_c more and more accurately.

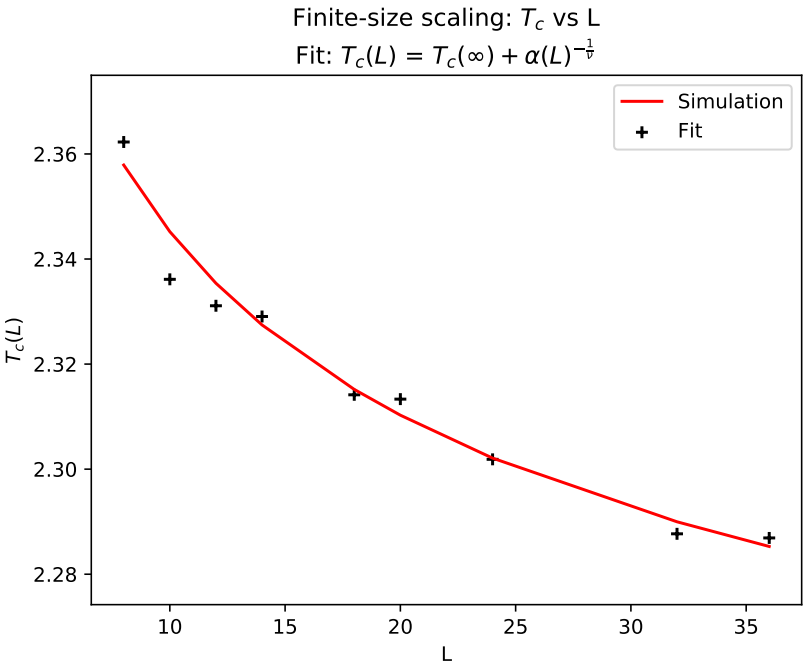


Figure 10: Both the simulation and the fit are seen to converge to Onsager's result at 2.26 J/k_B.

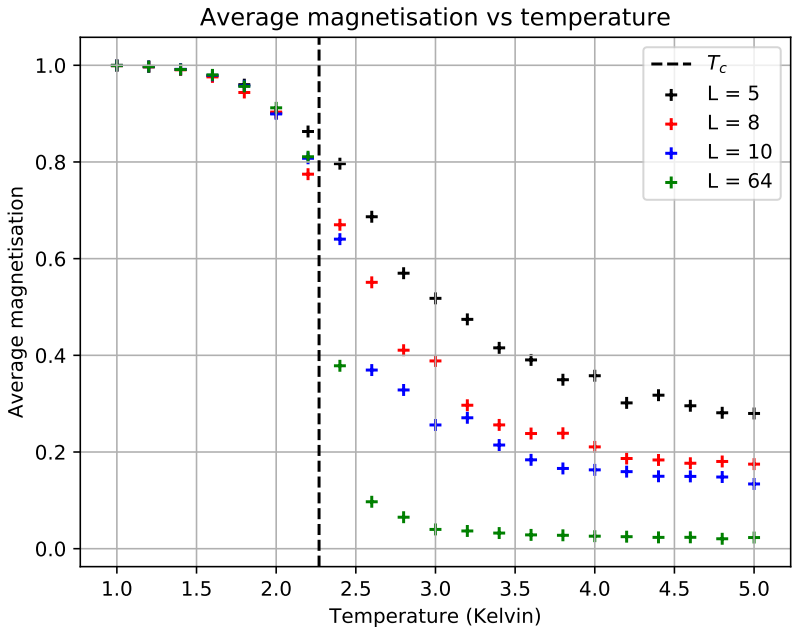


Figure 11: As L is increased, the behaviour approaches the analytical result that predicts the phase transition at $T_c = \frac{2}{ln(1+\sqrt{2})}$.

3.4 Formation of clusters around T_c

Code used: domains.py

3.4.1 Theory & Methods

Large magnetic domains form near T_c where the correlation length $\xi \to \infty$. Above T_c the clusters should dissipate due to the high probability of spin flips and below T_c the typical cluster size $\to$ N, the total number of sites.

For this investigation breadth-first-search (BFS) was used with a Queue data structure for $\mathcal{O}(1)$ complexity to adding and deleting the elements used. The time complexity of BFS is $\mathcal{O}(N+E)$, where N is the total number of sites and E are the edges connecting neighbours with identical spin. The typical cluster size was taken to be the average of all the cluster sizes above half of the maximum size.

3.4.2 Results

Figure 12 shows that, near the critical temperature clusters stop reducing in size and there is some increase as well. As $T \to \infty$ the cluster size $\to 1$. The

error is larger around T_c since there is a bigger range of configurations that the system can take at that temperature.

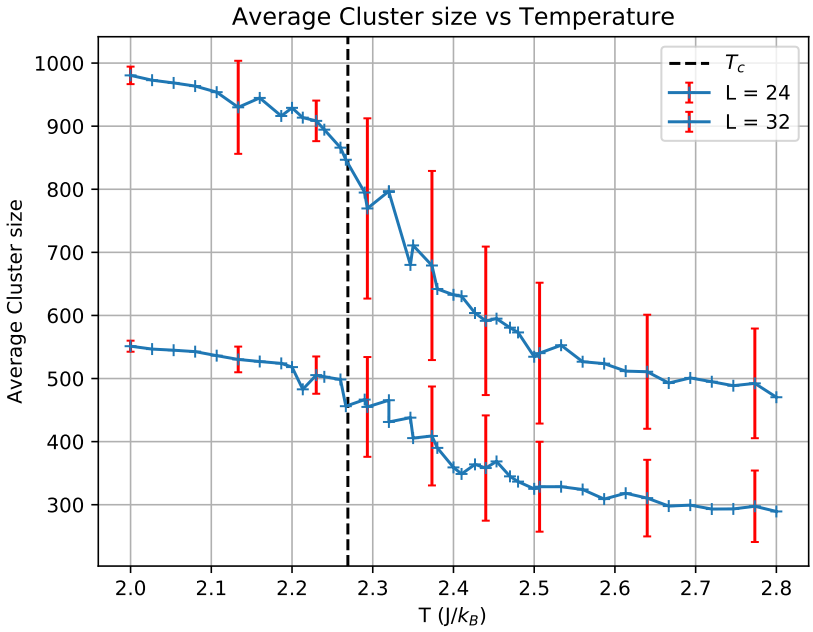


Figure 12: Top line has $L = 32$ and bottom line has $L = 24$. Around T_c, clusters stop dissipating as T increases and the errors are largest due to the large number of possible configurations.

3.5 Hysteresis and meta-stable states

Code used: hysteresis.py

3.5.1 Theory & Methods

A first order phase transition is characterised by the energy being discontinuous with respect to the order parameter below the critical temperature and continues above it [8]. Hysteresis which affects ferromagnetic materials - which the Ising Model represents - manifests below the critical temperature and is characterised by remnant magnetisation remaining when the applied magnetic field is removed. The material will reach zero magnetisation when a field is applied in the opposite direction, called the coercive field. When the applied field H is cycled, the average magnetisation vs H plot forms a hysteresis loop [9].

To investigate the above, a sample of $L = 10$ was cycled in H at different temperatures and plots of the average magnetisation and energy were made against H.

3.5.2 Results

From Figure 13 we observe that below T_c, magnetisation is discontinuous and forms a hysteresis loop. As T is increased, the transition from all spins aligned up to all spins aligned down becomes smoother and at large T, average magnetisation is linearly proportional to H as the material acts as a paramagnet.

In Figure 14 the energy is seen to be non-smooth below T_c as predicted and meta-stable states are shown. These states give an X-shape to the energy and emanate from the memory property of hysteresis. Further, as T is increased above T_c, the energy becomes smooth and continuous with no meta-stable states as no hysteresis occurs.

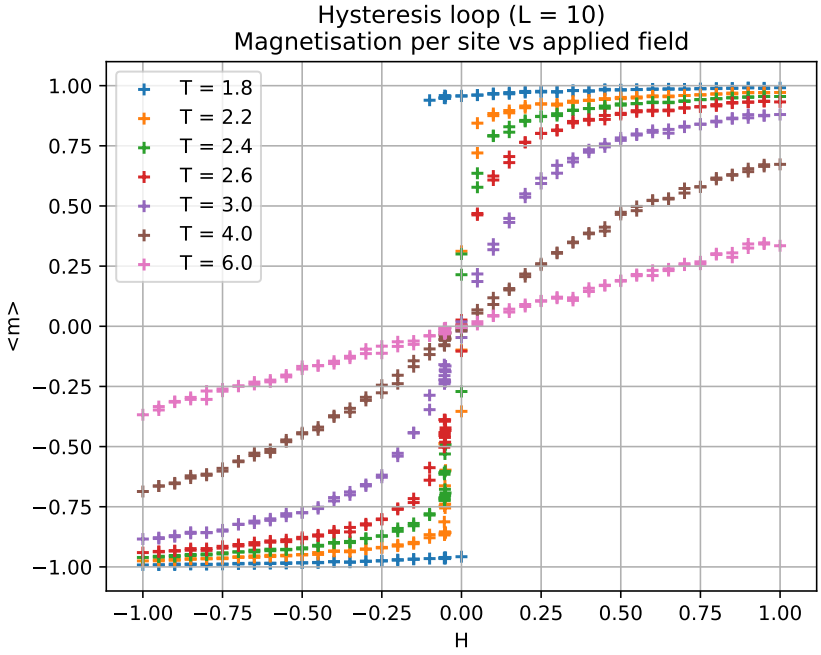


Figure 13: Average magnetisation vs H at different temperatures below and above T_c. Paramagnetism occurs at large $T > T_c$ and hysteresis loops form below T_c.

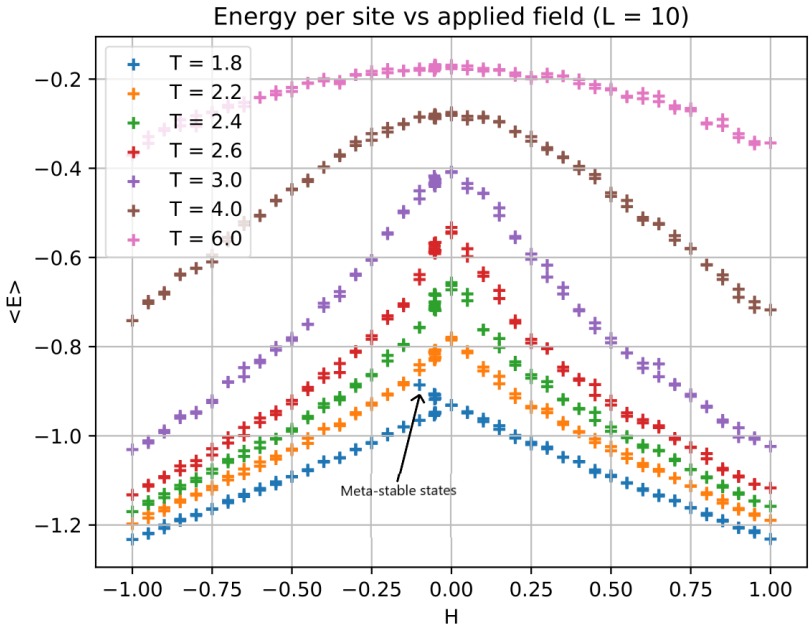


Figure 14: Average energy vs H, from a non-smooth function below T_c with meta-stable states, to a continuous function above T_c with no meta-stable states.

3.6 Magnetic susceptibility using bootstrap

Code used: susceptibility.py

3.6.1 Theory & Methods

The magnetic susceptibility is given by [3]

$$\chi = \frac{\sigma_M^2}{k_B T^2} \tag{8}$$

where σ_M is the standard deviation of the average magnetisation derived from the bootstrap method described in section 3.2.1. Similarly, the error in χ was found using $\sigma_\chi = (1 + 2\tau_e)\sigma_{n_b}$.

3.6.2 Results

In Figure 15 around T_c, χ takes its maximum value with its biggest error since at that temperature magnetisation changes rapidly. It is also evident that as $T \to 0, \infty$, $\chi \to 0$. This is because at low T all spins are aligned and the system is ferromagnetic, hence magnetisation is not changing and $\sigma_M \to 0$. At large T,

each site has a 50% chance of being spin up or down and magnetisation again stays constant around zero.

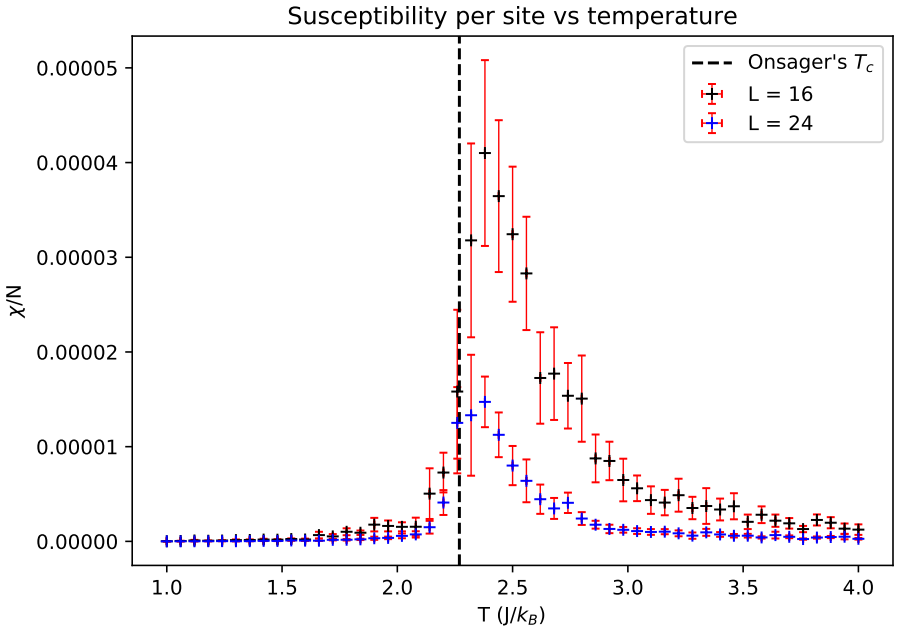


Figure 15: Susceptibility per site vs temperature for different lattice dimensions. As L is increased, susceptibility takes smaller values and the peak approaches Onsager's analytical T_c.

3.7 Next-nearest neighbour interactions

Code used: nearest_neighbours.py

3.7.1 Theory & Methods

This section is concerned with the effect of adding more interactions for each spin site. Concretely, in a 2D lattice the neighbours are left, right, forward, backward with the exchange energy constant at $J_0 = 1$ and the 4 diagonal next-nearest neighbours with the exchange energy J varied for different lattice dimensions. The following Hamiltonian describes the above:

$$H_{int} = -J_0 \sum_{\langle ij \rangle} s_i s_j - J \sum_{\langle ik \rangle} s_i s_k - \mu H \sum_{i=1}^{N} s_i \ . \tag{9}$$

By adding more interactions we approach a more realistic model, at the expense of increasing the time complexity of the simulation. A plot of average

magnetisation vs temperature is made for different values of J to observe where the spontaneous magnetisation is lost and hence qualitatively understand how T_c is affected.

3.7.2 Results

The plots in Figures 16 and 17 include Onsager's analytical T_c which points out that with more interactions the critical temperature increases. As expected, the model returns to the original nearest neighbours model as $J \to 0$ and as L is increased the transition becomes sharper and well-defined.

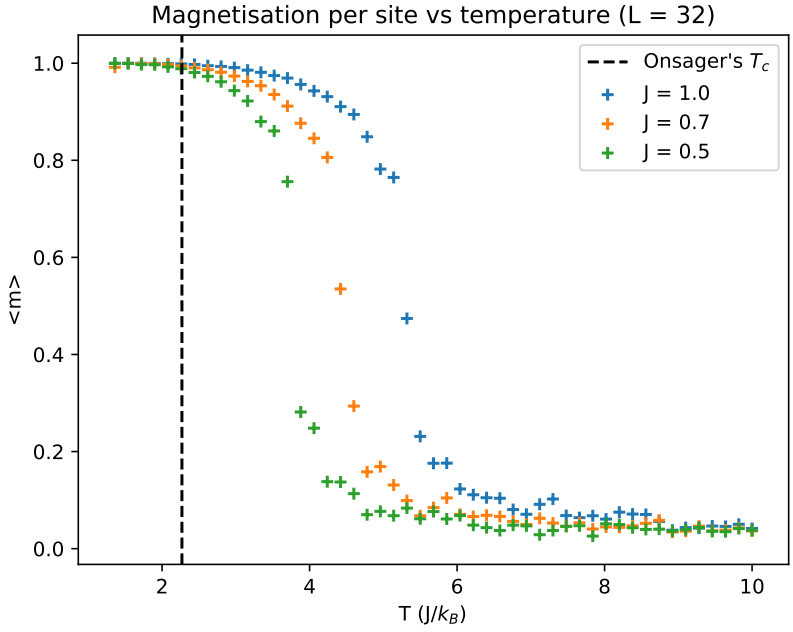


Figure 16: Next-nearest neighbour interactions cause T_c to increase and varies with the degree of the exchange energy magnitude. The dashed line is the critical temperature when next-nearest neighbour interactions are turned off.

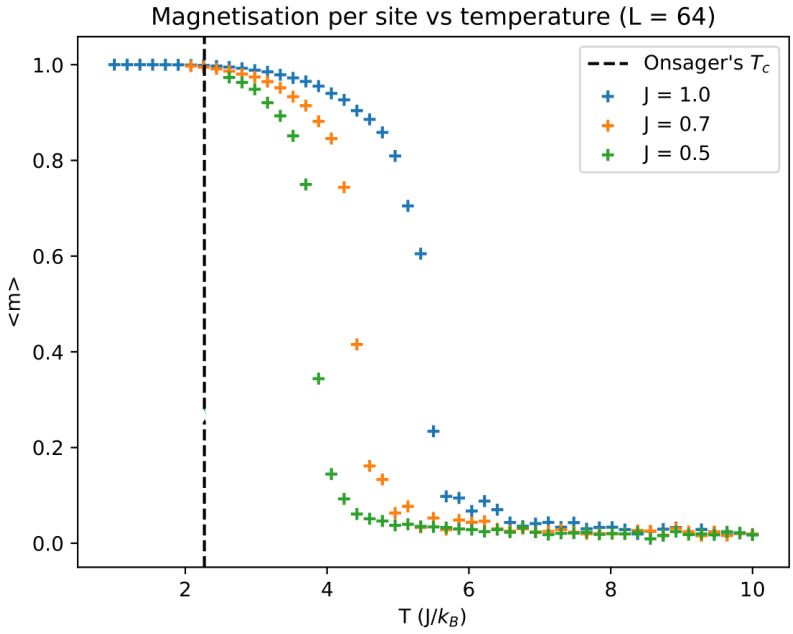


Figure 17: The next-nearest neighbour phase transition becomes sharper as $L \to \infty$. For $J = 1$, T_c is around 5.5 J/k_B, which is more than double from the model investigated in the previous sections, shown here with the dashed line.

3.8 Three-dimensional cubic lattice

Code used: 3d_lattice.py

3.8.1 Theory & Methods

For the 3D case there is no analytical solution for the critical temperature. It is a more realistic model, but again at the expense of higher time complexity.

The method for investigating average magnetisation vs temperature was the same as in section 3.3.1 but with the neighbours of a site now including an up and down giving a total of six neighbours.

3.8.2 Results

The critical temperature was found around 4.5 J/k_B as depicted in Figures 18 and 19. The average magnetisation goes to zero faster after T_c as we increase the lattice dimension.

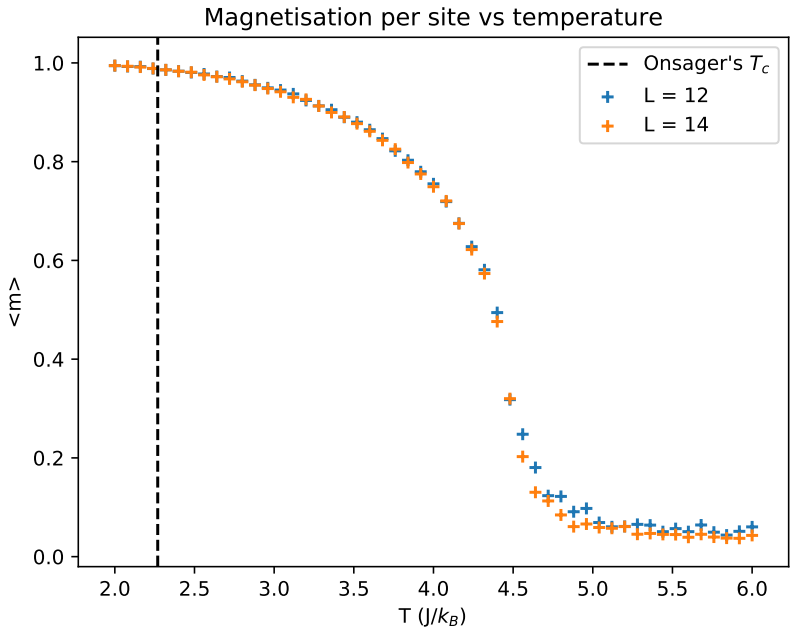


Figure 18: Average magnetisation vs temperature for the 3D Ising model, showing a T_c around 4.5 J/k_B. The dashed line is the critical temperature in 2D.

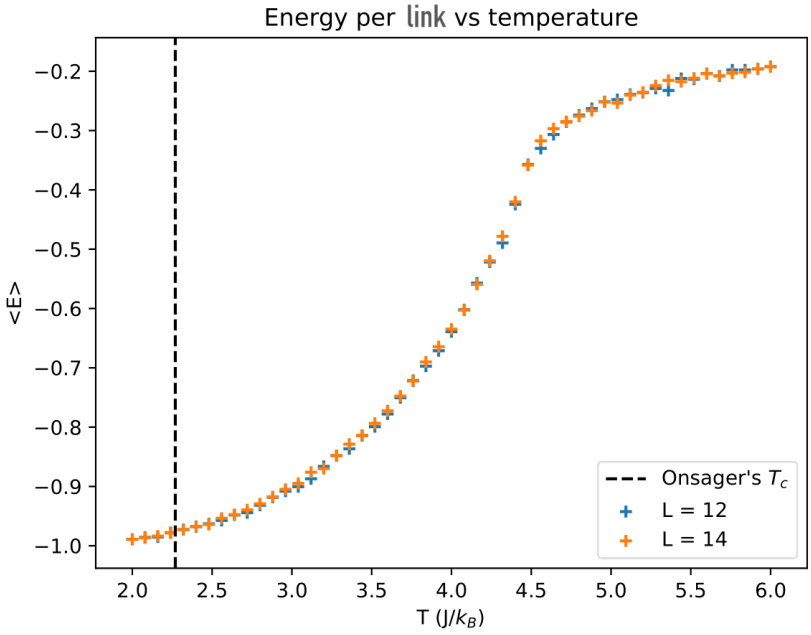


Figure 19: The energy vs temperature for the 3D case again indicates a higher T_c and as $T \to \infty$, energy $\to 0$, since at high T it is more likely that half of the neighbours are aligned and the other half are anti-aligned.

3.9 Supervised learning with a DNN

Code used: deep_ising_temp.py

3.9.1 Theory & Methods

A shallow neural network [10], multi-layered perceptron (MLP), has multiple layers including an input and an output layer. The layers are made of nodes and the nodes are connected with edges that carry weights. The weights are to be optimised using back-propagation so that the error on predictions for the labeled data (supervised learning) is minimised. For more details on neural networks the reader is referred to [10].

The MLP used here takes a 2D Ising lattice configuration as input and outputs the temperature for that state. The architecture was L×L nodes for the input layer - where L is the lattice dimension (L = 10), L nodes for the next hidden layer and 1 node for the output layer that would give the temperature prediction. No dropout (regularization) was used. The activation function for the hidden layer was a sigmoid and a rectified linear unit (relu) for the output layer. The optimisation method used was 'Adam' with the metric being the mean squared error (MSE).

After training the model, 5000 thermalised configurations were prepared for a given temperature and passed into the network for it to make a temperature prediction. A histogram of its predictions was plotted with a vertical line showing the temperature set for the configurations to thermalise. A second plot was produced to show the training error with the validation error - the validation set being 20% of the 50 000 lattice configurations thermalised at random temperatures between 1.8 and 6 as data for the network. The latter plot was used as an indication to overfitting.

3.9.2 Results

The training error achieved was quite low at around 0.3 MSE, however the validation error seemed to increase rapidly and a cut-off at 20 epochs gave a validation error 0.6 MSE. This showed that the network wouldn't generalise as well as expected to unseen configurations. Nevertheless, once given 5000 data of the same temperature to make predictions on, it was seen to follow the central limit theorem [11] and produce a histogram gaussian with a mean at the correct temperature. The following figures summarise the above.

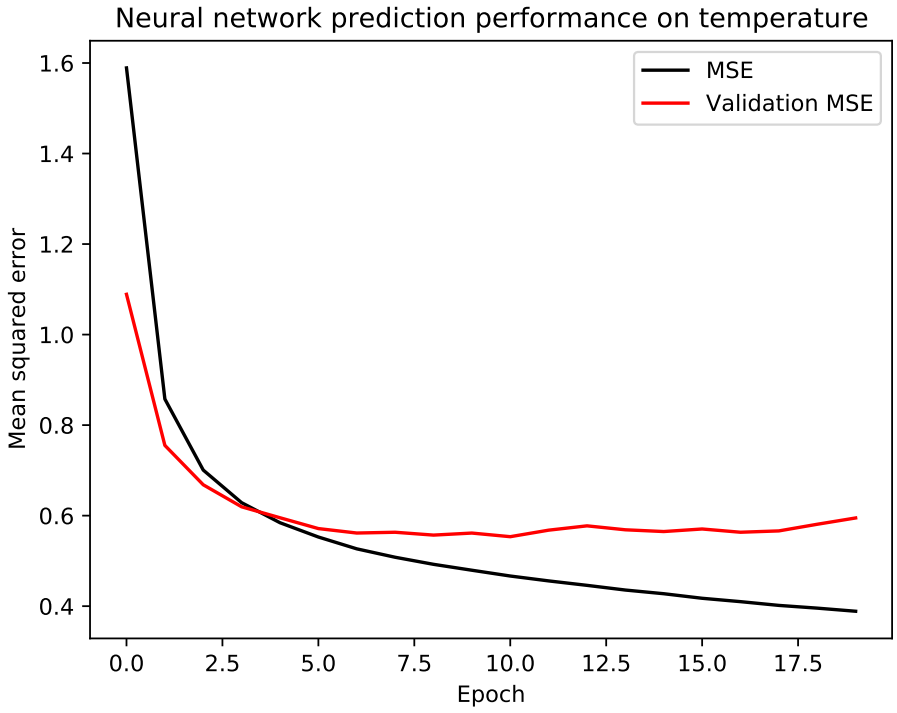


Figure 20: After 5 epochs, the validation starts diverging which is an indication of over-fitting.

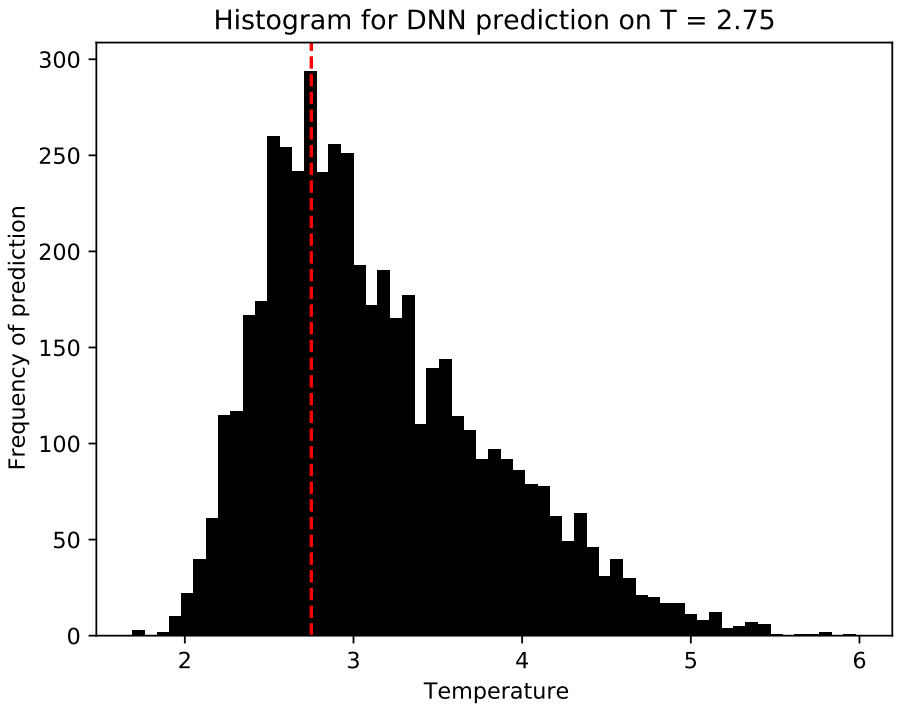


Figure 21: The prediction on 5000 configurations all with T = 2.75 shows gaussian behaviour as suggested by the central limit theorem with a mean at the correct temperature. The red dashed line is at T = 2.75.

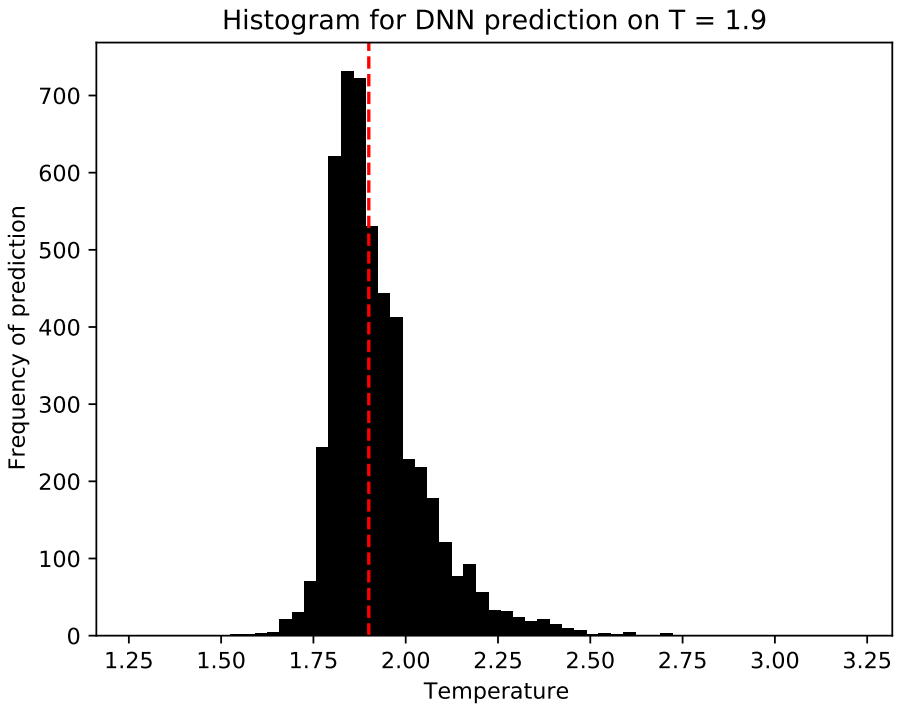


Figure 22: The prediction on 5000 configurations all with T = 1.9 shows gaussian behaviour as suggested by the central limit theorem with a mean at the correct temperature. Below T_c, all configurations are almost identical - hence the spread is less than in Figure 21 - and the mean is again centered on the correct temperature. It is possible that the model has picked up and learned on small fluctuations occuring at the higher temperatures below T_c. The red dashed line is at T = 1.9

4 Discussion

Some subjects required more computational time than others. For example the heat capacity which includes the bootstrap method was expensive to plot against temperature and small L values were used. With more computational power and bigger L, the plot can be made more smooth and sharp so that T_c is found with a smaller error.

The finite size scaling fit could have been improved with the evaluation of points at higher L. However, data points were expensive to compute and with more power a better estimate of $T_c(\infty)$ can be found. For all the subjects simulated, it was the case that higher lattice dimensions gave clearer, sharper results that were closest to theoretical predictions.

It was evident that the bigger the number of sweeps used, the more accurate

the results were due to allowing the system to thermalise completely and get more data to average over. However, more sweeps increase space and time complexity sometimes to non-tolerable levels. Also, in some cases the change was too fast to capture with more points as for example in Figure 16.

The performance of the neural network could easily be improved using Bayesian hyper-parameter optimisation which uses exploration vs exploitation to tackle the curse of dimensionality in the hyper-parameter space. It is the most efficient way to configure the architecture, especially for large and complicated models. It is not surprising that the model had trouble predicting the temperature some of the time and hence the spread in Figure 21, due to the same configuration being valid for different temperatures. However, different machine learning models such as Restricted Boltzmann Machines (RBM) perform better in producing observables, especially with deep architectures [12].

5 Conclusion

This paper on the Ising model included computational investigations on 2D, 3D lattices with next-nearest neighbour interactions, hysteresis, the behaviour of heat capacity and susceptibility with temperature, finite size scaling critical temperature extrapolation; with Gaussian curve fitting on the heat capacity curve and the evaluation of errors on observables using the bootstrap method. A neural network was also trained with supervised learning to predict the temperature of a given configuration. The following points summarise the findings of this paper:

1. Away from T_c, magnetisation fluctuations go to 0.

2. The time constant for decorrelation grows with the lattice dimension and is biggest at T_c. The z exponent in the critical slowing down relationship was found to be $z = 2.167 \pm 0.037$.

3. Heat capacity and susceptibility were largest at T_c and decayed to zero away from T_c. The best estimate for the critical temperature found using a Gaussian fit on the heat capacity plot was $T_c = 2.2758 \pm 0.0096\ J/k_B$. Finite size scaling gave $T_c(\infty) = 2.264 \pm 0.152\ J/k_B$.

4. The average magnetisation vs temperature plot shows spontaneous magnetisation at and below the critical temperature. The latter is the manifestation of a broken symmetry where the order parameter has an unstable equilibrium at 0 magnetisation and hence the system spontaneously acquires a new ground state at an average magnetisation of 1 or -1.

5. Near T_c large clusters were found to form and as $T \to \infty$ the average cluster size decayed to one. Above T_c there is no hysteresis and no metastable states and vice versa. As $T \to \infty$, average magnetisation was proportional to the applied magnetic field.

6. Next-nearest neighbour interactions with exchange energy $J = 1$ gave $T_c \approx 5.5\ J/k_B$. The 3D cubic lattice had a critical temperature of phase transition at approximately $4.5\ J/k_B$.

7. Supervised learning of the neural network to predict the temperature of a given 2D configuration performed well in the limit of many predictions on a single temperature.

Word count: ∼2500

References

[1] Martin Niss. *History of the Lenz-Ising Model 1920–1950: From Ferromagnetic to Cooperative Phenomena.* Archive for History of Exact Sciences, 2004.

[2] Dong Zhang Jia-Ji Zhu Kai Chang Rui Zhang, Bin Wei. *Few-shot machine learning in the three-dimensional Ising model.* Phys. Rev. B 99, 094427(2019), 2019.

[3] Konstantinos Anagnostopoulos. *Computational Physics: A Practical Introduction to Computational Physics and Scientific Computing (using C++).* National Technical University of Athens, 2016.

[4] François Chollet et al. Keras. https://keras.io, 2015.

[5] *A New Kind of Science.* Wolfram Media Inc., 2002.

[6] Jean-Dominique Deuschel Amir Dembo. *Markovian perturbation, response and fluctuation dissipation theorem.* arXiv:0710.4394, 2007.

[7] K. Binder. *Finite size scaling analysis of ising model block distribution functions.* Zeitschrift für Physik B Condensed Matter, 1981.

[8] Yeomans J. M. *Statistical Mechanics of Phase Transitions.* Oxford University Press, 1992.

[9] Carl W. Garland and Rémi Renard. *Order—Disorder Phenomena. I. Instability and Hysteresis in an Ising Model Near Its Critical Point.* The Journal of Chemical Physics, 2004.

[10] Liu W. et al. *A survey of deep neural network architectures and their applications.* Neurocomputing, 2017.

[11] Richard Durrett. *Probability: theory and examples (3rd ed.).* Cambridge University Press, 2004.

[12] Alan Morningstar and Roger G. Melko. Deep learning the ising model near criticality. *J. Mach. Learn. Res.*, 2017.

6 Code

Some code files used for plotting are not included in this code listing.

Listings

Code file: avg_mag.py

```
from random import uniform, randint
import random
from math import exp
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Global variables
L = 10 # Lattice dimension
N = L**2 # Total number of nodes in lattice
J = 1 # Exchange energy
H = -2.5 # Applied magnetic field strength
mu = 1 # Magnetic moment
M = 0 # Total magnetisation
T = 0.9 # Temperature in Kelvin
b = 1/T # Constant: 1 / temperature
dE_4 = exp(-4*b) # Probability for energy change of +4
dE_8 = exp(-8*b) # Probability for energy change of +8
number_of_neighbours = 4
random_seed = 31 # Fix random seed in spin initialization for reproducability
hot_start = True # Initialize with hot start
nsweeps = 2000 # Number of sweeps

def get_spin_list(is_hot):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make
     cold start
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(N):
```

```
            rdm_num = uniform(0, 1) # Get a float between 0 and 1 from a uniform distribution
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N # Creates a list of N (+1)'s
    return s_local

def get_neighbours_index():
    """
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {} # Index of site -> indexes of neighbours of that site (key -> value)
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
            bottom = i + L - N
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

s = get_spin_list(hot_start) # Initialise a lattice
neighbours_dictionary = get_neighbours_index() # Initialise the dictionary holding the neighbours

def get_energy_difference(index_to_flip):
    """
    :param index_to_flip: (int) the site index to consider flipping its spin
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours + 2*s[index_to_flip]*mu*H
    return total_change

def metropolis():
    """
    The metropolis algorithm as a markov chain monte carlo simulation algorithm that modifies the spin state of the
    lattice and gives a new state by choosing N (= L x L) sites at random and checking through the energy if it will
    flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
```

```
    spin will be flipped and the numbers 4, 8
    represent the corresponding change in energy so that we don't calculate
    many times an exponential term
    :return: (void) does not return anything but changes the state of s
    """
    for i in range(N):
        site_index = randint(0, N-1)
        dE = get_energy_difference(site_index)
        rdm_num = uniform(0, 1)
        if dE <= 0:
            s[site_index] *= -1
        elif dE == 4:
            if dE_4 > rdm_num:
                s[site_index] *= -1
            else:
                continue
        elif dE == 8:
            if dE_8 > rdm_num:
                s[site_index] *= -1
            else:
                continue

def get_magnetisation():
    """
    :return: (float) Total magnetisation
    """
    magnetisation_total = 0
    for i in range(N):
        magnetisation_total += s[i]
    return magnetisation_total

def get_energy():
    """
    :return: (float) Total energy through the Hamiltonian
    """
    sum1 = 0
    sum2 = 0
    for i in range(N):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation()
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy

def get_average_energy():
    """
    :return: (float) Average energy through the Hamiltonian
    """
    sum1 = 0
    sum2 = 0
    for i in range(N):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation()
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy/(2*N)

def get_average_magnetisation():
    """
    :return: (float) Magnetisation per site
    """
    return get_magnetisation()/N

def simulation():
    """
    Goes through all sweeps modifying the state with the metropolis function
```

```
    and logging total
    magnetisation and energy. Then calculates the mean and standard deviation
     of the total magnetisation after
    thermalisation to quantify its fluctuations. The it plots time vs total
    and average energy and time vs magnetisation
    :return: (void)
    """
    total_magnetisation = []
    total_energy = []
    nsweeps_values = np.arange(0, nsweeps)

    # Evolve the system and save magnetisation and energy for each new state
    for i in range(nsweeps):
        metropolis()
        total_magnetisation.append(get_magnetisation())
        total_energy.append(get_energy())

    # Quantify fluctuations after thermalisation of 1000 sweeps
    mean = np.average(total_magnetisation[1000:])
    standard_deviation = np.std(total_magnetisation[1000:])

    plt.plot(nsweeps_values, total_magnetisation, color = 'k')
    plt.xlabel('Time (sweeps)')
    plt.ylabel('Total magnetisation')
    plt.title(f'Time vs total magnetisation\nThermalised mean: {mean:.3f},
    Std: {standard_deviation:.3f}\nTemperature: {T}K, L = {L}')
    plt.grid(True)
    # plt.savefig(f'time_vs_total_mag_{T}.pdf', bbox = 'tight')
    plt.show()

    plt.plot(nsweeps_values, np.array(total_magnetisation)/N, color='k')
    plt.xlabel('Time (sweeps)')
    plt.ylabel('Average magnetisation')
    plt.title(
        f'Time vs average magnetisation\nTemperature: {T}K, L = {L}')
    plt.grid(True)
    # plt.savefig(f't_vs_avg_mag_{T}.pdf', bbox = 'tight')
    plt.show()

    plt.plot(nsweeps_values, total_energy, color = 'k')
    plt.xlabel('Time (sweeps)')
    plt.ylabel('Total energy')
    plt.title(f'Time vs total energy\nTemperature: {T}K, L = {L}')
    plt.grid(True)
    # plt.savefig(f'time_vs_total_energy_{T}.pdf', bbox = 'tight')
    plt.show()

    average_energy = np.array(total_energy)/(2*N)
    plt.plot(nsweeps_values, average_energy, color='k')
    plt.xlabel('Time (sweeps)')
    plt.ylabel('Average energy')
    plt.title(f'Time vs average energy\nTemperature: {T}K, L = {L}')
    plt.grid(True)
    # plt.savefig(f'time_vs_avg_energy_{T}.pdf', bbox = 'tight')
    plt.show()

simulation()
```

Code file: ising_ca_reader.py

```
import pyglet
import numpy as np

class Lattice:

    def __init__(self, window_width, window_height, cell_size):
        self.grid_width = int(window_width / cell_size)
        self.grid_height = int(window_height / cell_size)
```

```
        self.cell_size = cell_size
        self.configurations = self.get_sites()
        self.config_num = 0

    def get_sites(self):
        with open('ising_ca.txt', 'r') as file:
            configurations = []
            line = file.readline()
            while line:
                configuration = line.strip().split(' ')
                configurations.append(configuration)
                line = file.readline()
        return configurations

    def draw(self):
        try:
            configuration = self.configurations[self.config_num]
        except IndexError:
            print('The simulation is over!')
            quit()
        L = int(np.sqrt(len(configuration)))
        for i in range(len(configuration)):
            row = int(i/L)
            col = i % L
            if int(configuration[i]) == 1:
                square_coords = (row*self.cell_size, col*self.cell_size,
                                 row*self.cell_size, col*self.cell_size +
    self.cell_size,
                                 row*self.cell_size + self.cell_size, col*
    self.cell_size,
                                 row*self.cell_size+self.cell_size, col*self.
    cell_size+self.cell_size)
                pyglet.graphics.draw_indexed(4, pyglet.gl.GL_TRIANGLES, [0,
    1, 2, 1, 2, 3], ('v2i', square_coords))

class Window(pyglet.window.Window):

    def __init__(self):
        super().__init__(600, 600)
        self.lattice = Lattice(self.get_size()[0], self.get_size()[1], int(
    self.get_size()[0]/50))
        pyglet.clock.schedule_interval(self.update, 1.0/25.0)

    def on_draw(self):
        self.clear()
        self.lattice.draw()

    def update(self, dt):
        self.lattice.config_num += 1

if __name__ == '__main__':
    window = Window()
    pyglet.app.run()
```

Code file: avg_mag_table.py

```
from random import uniform, randint
import random
from math import exp
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Global variables
J = 1 # Exchange energy
H = 0 # Applied magnetic field strength
mu = 1 # Magnetic moment
M = 0 # Total magnetisation
```

```
number_of_neighbours = 4
random_seed = 10 # Fix random seed in spin initialization for reproducability
hot_start = False # Initialize with hot start or not
nsweeps = 1000 # Number of sweeps

def get_spin_list(is_hot, N):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make cold start
    :param N: (int) N = L x L, total number of sites
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(N):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N
    return s_local

def get_neighbours_index(N):
    """
    :param N: (int) N = L x L, total number of sites
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {}
    L = int(N**(1/2))
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
            bottom = i + L - N
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

def get_energy_difference(index_to_flip, s, neighbours_dictionary):
    """
    :param index_to_flip: (int) the site index to consider flipping its spin
    :param s: (list) spin list of the lattice
    :param neighbours_dictionary: (dict) holds indexes for each site's neighbours
```

```
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours # Works out from the Hamiltonian of the before and after states
    return total_change

def metropolis(dE_4, dE_8, N, s, neighbours_dictionary):
    """
    The metropolis algorithm as a markov chain monte carlo simulation algorithm that modifies the spin state of the
    lattice and gives a new state by choosing N (= L x L) sites at random and checking through the energy if it will
    flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the spin will be flipped and the numbers 4, 8
    represent the corresponding change in energy so that we don't calculate many times an exponential term
    :param dE_4: (float) Probability for energy change of +4
    :param dE_8: (float) Probability for energy change of +8
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (void) does not return anything but changes the state of s
    """
    for i in range(N):
        site_index = randint(0,N-1)
        dE = get_energy_difference(site_index, s, neighbours_dictionary)
        rdm_num = uniform(0, 1)
        if dE <= 0:
            s[site_index] *= -1
        elif dE == 4:
            if dE_4 > rdm_num:
                s[site_index] *= -1
            else:
                continue
        elif dE == 8:
            if dE_8 > rdm_num:
                s[site_index] *= -1
            else:
                continue

def get_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Total magnetisation
    """
    magnetisation_total = 0
    for i in range(N):
        magnetisation_total += s[i]
    return magnetisation_total

def get_energy(N, s, neighbours_dictionary):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (float) Total energy through the Hamiltonian
    """
    sum1 = 0
    sum2 = 0
    for i in range(N):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation(N, s)
```

```
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy

def get_average_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Magnetisation per site
    """
    return abs(get_magnetisation(N, s))/N

def simulation():
    """
    The simulation function gathers data for average magnetisation vs temperature for different L dimensions of the
    lattice. It stores the data in a pandas dataframe and makes a plot of all L in the same average magnetisation vs
    temperature graph
    :return: (void)
    """
    # Temperature values in Kelvin
    T_values = np.linspace(1, 5, num = 21)
    # Dimension values
    L_values = [5, 8, 10, 64]
    # Dataframe to store magnetisation values vs temperature for different L
    dataframe = pd.DataFrame() # T_values, columns = ['Temperature']
    for L in L_values:
        N = L**2
        s = get_spin_list(hot_start, N)
        neighbours_dictionary = get_neighbours_index(N)
        average_magnetisation = []
        for T in T_values:
            b = 1 / T  # Constant: 1 / temperature
            dE_4 = exp(-4 * b)  # Probability for energy change of +4
            dE_8 = exp(-8 * b)  # Probability for energy change of +8
            tmp_magnetisation = []
            for i in range(nsweeps):
                metropolis(dE_4, dE_8, N, s, neighbours_dictionary)
                tmp_magnetisation.append(get_average_magnetisation(N, s))
            average_magnetisation.append(np.average(tmp_magnetisation[100:]))
            print(T)
        dataframe[f'L{L}'] = average_magnetisation
        print(L, '--------------------')

    plt.scatter(T_values, dataframe['L5'], color = 'k', marker = '+')
    plt.scatter(T_values, dataframe['L8'], color = 'r', marker = '+')
    plt.scatter(T_values, dataframe['L10'], color = 'b', marker = '+')
    plt.scatter(T_values, dataframe['L64'], color = 'g', marker = '+')
    plt.axvline(x = 2/np.log(1+np.sqrt(2)), color = 'k', linestyle = '--')
    plt.xlabel('T (J/$k_B$)')
    plt.ylabel('Average magnetisation')
    plt.title('Average magnetisation vs temperature')
    plt.grid(True)
    plt.legend(['$T_c$', 'L = 5', 'L = 8', 'L = 10', 'L = 64'])
    # plt.savefig('avg_mag_vs_temp.pdf', bbox = 'tight')
    plt.show()

simulation()
```

Code file: bootstrap.py

```
import random
import numpy as np

def get_sigma_energy(pseudo_bin_list):
    """
    :param pseudo_bin_list: (list) contains the pseudo data of one bin
    :return: (float) the standard deviation of the input list
```

```
    """
    return np.std(pseudo_bin_list)

def get_heat_capacity(sigma_energy, T):
    """
    :param sigma_energy: (float) the standard deviation of the energy from a bin list
    :param T: (float) Temperature
    :return: (float) the specific heat capacity
    """
    return (sigma_energy**2) / (T**2)

def bootstrap(energy_list, n_bins, T, target_tau):
    """
    :param target_tau: (int) decorrelation constant
    :param energy_list: (list) energy for each sweep
    :param n_bins: (int) number of bins to include in the boostrap method
    :param T: (float) Temperature
    :return: (float, float) 2-tuple of heat capacity and its standard deviation
    """
    length = len(energy_list)
    # Initialise the bin dictionary holding values of lists. These lists will contain pseudo data of energy
    bins_dict = {idx: [] for idx in range(n_bins)}
    # This for loop traverses the bin keys in the dictionary bins_dict
    for bin_index in range(n_bins):
        # This for loop runs for the length of the energy list
        for i in range(length):
            random_index = random.randint(0, length-1)
            bins_dict[bin_index].append(energy_list[random_index]) # Append pseudo data to bin list

    sigma_energy_list = [get_sigma_energy(bins_dict[key_bin]) for key_bin in bins_dict]
    C_list = [get_heat_capacity(sigma_energy_list[i], T) for i in range(len(sigma_energy_list))]

    C_mean = np.average(C_list)
    sigma_C = np.std(C_list)*np.sqrt(1 + 2*target_tau)

    return C_mean, sigma_C
```

Code file: bootstrap_susceptibility.py

```
import random
import numpy as np

def get_sigma_magnetisation(pseudo_bin_list):
    """
    :param pseudo_bin_list: (list) contains the pseudo data of one bin
    :return: (float) the standard deviation of the input list
    """
    return np.std(pseudo_bin_list)

def get_susceptibility(sigma_magnetisation, T):
    """
    :param sigma_magnetisation: (float) the standard deviation of the magnetisation from a bin list
    :param T: (float) Temperature
    :return: (float) the susceptibility
    """
    return (sigma_magnetisation**2) / (T**2)

def bootstrap(magnetisation_list, n_bins, T, target_tau):
    """
    :param target_tau: (int) decorrelation constant
    :param magnetisation_list: (list) magnetisation for each sweep
```

```
    :param n_bins: (int) number of bins to include in the boostrap method
    :param T: (float) Temperature
    :return: (float, float) 2-tuple of susceptibility and its standard deviation
    """
    length = len(magnetisation_list)
    # Initialise the bin dictionary holding values of lists. These lists will contain pseudo data of magnetisation
    bins_dict = {idx: [] for idx in range(n_bins)}
    # This for loop traverses the bin keys in the dictionary bins_dict
    for bin_index in range(n_bins):
        # This for loop runs for the length of the magnetisation list
        for i in range(length):
            random_index = random.randint(0, length-1)
            bins_dict[bin_index].append(magnetisation_list[random_index]) # Append pseudo data to bin list

    sigma_magnetisation_list = [get_sigma_magnetisation(bins_dict[key_bin]) for key_bin in bins_dict]
    chi_list = [get_susceptibility(sigma_magnetisation_list[i], T) for i in range(len(sigma_magnetisation_list))]

    chi_mean = np.average(chi_list)
    sigma_chi = np.std(chi_list)*np.sqrt(1 + 2*target_tau)

    return chi_mean, sigma_chi
```

Code file: correlation.py

```
from random import uniform, randint
import random
from math import exp
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Global variables
L = 30 # Lattice dimension
N = L**2 # Total number of nodes in lattice
J = 1 # Exchange energy
H = 0 # Applied magnetic field strength
mu = 1 # Magnetic moment
M = 0 # Total magnetisation
T = 1.0 # Temperature in Kelvin
b = 1/T # Constant: 1 / temperature
dE_4 = exp(-4*b) # Probability for energy change of +4
dE_8 = exp(-8*b) # Probability for energy change of +8
number_of_neighbours = 4
random_seed = 13 # Fix random seed in spin initialization for reproducability
hot_start = True # Initialize with hot start
nsweeps = 30000 # Number of sweeps

def get_spin_list(is_hot):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make cold start
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(N):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
```

```
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N
    return s_local

def get_neighbours_index():
    """
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {}
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
            bottom = i + L - N
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

s = get_spin_list(hot_start)
neighbours_dictionary = get_neighbours_index()

def get_energy_difference(index_to_flip):
    """
    :param index_to_flip: (int) the site index to consider flipping its spin
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours # Works out from the Hamiltonian of the before and after states
    return total_change

def metropolis():
    """
    The metropolis algorithm as a markov chain monte carlo simulation algorithm that modifies the spin state of the
    lattice and gives a new state by choosing N (= L x L) sites at random and checking through the energy if it will
    flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the spin will be flipped and the numbers 4, 8
    represent the corresponding change in energy so that we don't calculate many times an exponential term
    :return: (void) does not return anything but changes the state of s
    """
    for i in range(N):
        site_index = randint(0,N-1)
        dE = get_energy_difference(site_index)
```

```
        rdm_num = uniform(0, 1)
        if dE <= 0:
            s[site_index] *= -1
        elif dE == 4:
            if dE_4 > rdm_num:
                s[site_index] *= -1
            else:
                continue
        elif dE == 8:
            if dE_8 > rdm_num:
                s[site_index] *= -1
            else:
                continue

def get_magnetisation():
    """
    :return: (float) Total magnetisation
    """
    magnetisation_total = 0
    for i in range(N):
        magnetisation_total += s[i]
    return magnetisation_total

def get_energy():
    """
    :return: (float) Total energy through the Hamiltonian
    """
    sum1 = 0
    sum2 = 0
    for i in range(N):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation()
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy

def get_average_magnetisation():
    """
    :return: (float) Magnetisation per site
    """
    return abs(get_magnetisation())/N

def get_autocovariance(M_list, tau):
    """
    :param M_list: (list) holding average magnetisation for each sweep
     neglected pre-thermalised samples
    :param tau: (int) time lag which is the input to the autocovariance
     formula
    :return: (float) autocovariance for the time lag tau
    """
    mean = np.average(M_list)
    if len(M_list)-tau<=0:
        print(f'The length of M_list in auto_covariance is {len(M_list)} and
     of tau {tau}')
    autocovariance_list = [(M_list[t] - mean)*(M_list[t+tau] - mean) for t in
      range(len(M_list) - tau)]
    return  np.average(autocovariance_list)

def get_autocorrelation(M_list, tau):
    """
    :param M_list: (list) holding average magnetisation for each sweep
     neglected pre-thermalised samples
    :param tau: (int) time lag which is the input to the autocovariance
     formula
    :return: (float) autocorrelation for the time lag tau
    """
    A_0 = get_autocovariance(M_list, 0)
```

```
    A_tau = get_autocovariance(M_list, tau)
    return  A_tau/A_0

def get_target_value_index(autocorrelation_list, target_value):
    """
    When the autocorrelation falls below 1/e return the index
    :param autocorrelation_list: (list) autocorrelation for different tau values
    :param target_value: (float) 1/e of the initial autocorrelation value
    :return: (int) Index of tau that gives 1/e of initial autocorrelation
    """
    target_index = -1
    for i in range(len(autocorrelation_list)):
        if autocorrelation_list[i] < target_value:
            target_index = i
            break
        else:
            continue
    return target_index

def simulation():
    """
    Evolves the systems with nsweeps and records average magnetisation for every state. The the simulation
    finds the tau that makes autocorrelation fall to 1/e called the constant of decorrelation and plots
    autocorrelation vs tau
    :return: (void)
    """
    avg_magnetisation_list = []
    autocorrelation_list = []
    thermalisation_sweeps = 5000
    sample_every = 50

    for t in range(nsweeps):
        metropolis()
        if t < thermalisation_sweeps:
            continue
        else:
            if t % sample_every == 0:
                avg_magnetisation_list.append(get_average_magnetisation())

    tau_list = np.arange(0, 50, 1)

    for tau in tau_list:
        autocorrelation_list.append(get_autocorrelation(
    avg_magnetisation_list, tau))

    target_value = 1/np.e
    index = get_target_value_index(autocorrelation_list, target_value)
    target_tau = tau_list[index]

    plt.plot(tau_list, autocorrelation_list, color = 'k')
    plt.axhline(y = target_value, linestyle = '--', color = 'r')
    plt.axvline(x = target_tau, linestyle = '--', color='b')
    plt.title(f'Autocorrelation \u03b1(\u03C4) vs \u03C4\nTemperature: {T}K, L = {L}')
    plt.xlabel('\u03C4')
    plt.ylabel('Autocorrelation \u03b1(\u03C4)')
    plt.legend(['\u03b1(\u03C4)', '1/e', f'\u03C4 at 1/e = {target_tau}'])
    plt.grid(True)
    # plt.savefig(f'autocorrelation_vs_tau_{T}.pdf', bbox = 'tight')
    plt.show()

simulation()
```

Code file: correlation_table.py

```
from random import uniform, randint
import random
from math import exp
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.optimize import curve_fit
import time

# Global variables
J = 1 # Exchange energy
H = 0 # Applied magnetic field strength
mu = 1 # Magnetic moment
M = 0 # Total magnetisation
number_of_neighbours = 4
random_seed = 10 # Fix random seed in spin initialization for reproducability
hot_start = True # Initialize with hot start
nsweeps = 75000 # Number of sweeps

def get_spin_list(is_hot, N):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make cold start
    :param N: (int) N = L x L, total number of sites
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(N):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N
    return s_local

def get_neighbours_index(L):
    """
    :param L: (int) Dimension of the lattice
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {}
    N = L**2
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
```

```
            bottom = i + L - N
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

def get_energy_difference(s, neighbours_dictionary, index_to_flip):
    """
    :param s: (list) spin list of the lattice
    :param neighbours_dictionary: (dict) holds indexes for each site's neighbours
    :param index_to_flip: (int) the site index to consider flipping its spin
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours
    return total_change

def metropolis(s, neighbours_dictionary, N, dE_4, dE_8):
    """
    The metropolis algorithm as a markov chain monte carlo simulation algorithm that modifies the spin state of the
    lattice and gives a new state by choosing N (= L x L) sites at random and checking through the energy if it will
    flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the spin will be flipped and the numbers 4, 8
    represent the corresponding change in energy so that we don't calculate many times an exponential term
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :param N: (int) total number of sites
    :param dE_4: (float) Probability for energy change of +4
    :param dE_8: (float) Probability for energy change of +8
    :return: (void) does not return anything but changes the state of s
    """
    for i in range(N):
        site_index = randint(0,N-1)
        dE = get_energy_difference(s, neighbours_dictionary, site_index)
        rdm_num = uniform(0, 1)
        if dE <= 0:
            s[site_index] *= -1
        elif dE == 4:
            if dE_4 > rdm_num:
                s[site_index] *= -1
            else:
                continue
        elif dE == 8:
            if dE_8 > rdm_num:
                s[site_index] *= -1
            else:
                continue

def get_magnetisation(s, N):
    """
    :param s: (list) spin list
    :param N: (int) total number of sites
    :return: (float) Total magnetisation
    """
    magnetisation_total = 0
    for i in range(N):
        magnetisation_total += s[i]
    return magnetisation_total

def get_energy(N, s, neighbours_dictionary):
```

```
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (float) Total energy through the Hamiltonian
    """
    sum1 = 0
    sum2 = 0
    for i in range(N):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation(s, N)
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy

def get_average_magnetisation(s, N):
    """
    :param s: (list) spin list
    :param N: (int) total number of sites
    :return: (float) Magnetisation per site
    """
    return abs(get_magnetisation(s, N))/N

def get_autocovariance(M_list, tau):
    """
    :param M_list: (list) holding average magnetisation for each sweep
     neglected pre-thermalised samples
    :param tau: (int) time lag which is the input to the autocovariance
     formula
    :return: (float) autocovariance for the time lag tau
    """
    mean = np.average(M_list)
    autocovariance_list = [(M_list[t] - mean)*(M_list[t+tau] - mean) for t in
      range(len(M_list) - tau)]
    return  np.average(autocovariance_list)

def get_autocorrelation(M_list, tau):
    """
    :param M_list: (list) holding average magnetisation for each sweep
     neglected pre-thermalised samples
    :param tau: (int) time lag which is the input to the autocovariance
     formula
    :return: (float) autocorrelation for the time lag tau
    """
    A_0 = get_autocovariance(M_list, 0)
    A_tau = get_autocovariance(M_list, tau)
    return  A_tau/A_0

def get_target_value_index(autocorrelation_list, target_value):
    """
    :param autocorrelation_list: (list) autocorrelation for different tau
     values
    :param target_value: (float) autocorrelation initial value * 1/e
    :return: (int) index for target tau
    """
    target_index = -1
    for i in range(len(autocorrelation_list)):
        if autocorrelation_list[i] < target_value:
            target_index = i
            break
        else:
            continue
    return target_index

def get_target_tau(s, L, dE_4, dE_8, neighbours_dictionary):
    """
    :param s: (list) spin list
```

```
    :param L: (int) lattice dimension
    :param dE_4: (float) Probability for energy change of +4
    :param dE_8: (float) Probability for energy change of +8
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (int) tau that makes autocorrelation fall to 1/e
    """
    N = L**2
    avg_magnetisation_list = []
    autocorrelation_list = []
    tau_list = np.arange(0, 2500, 1)
    thermalisation_sweeps = 20000
    sample_every = 1

    for sweep in range(nsweeps):

        metropolis(s, neighbours_dictionary, N, dE_4, dE_8)

        if sweep < thermalisation_sweeps:
            continue
        else:
            if sweep % sample_every == 0:
                avg_magnetisation_list.append(get_average_magnetisation(s, N)
)
            else:
                continue

    for tau in tau_list:
        autocorrelation_list.append(get_autocorrelation(
avg_magnetisation_list, tau))

    target_value = 1/np.e
    index = get_target_value_index(autocorrelation_list, target_value)
    target_tau = tau_list[index]

    return target_tau

def simulation():
    """
    For different values of L and T the simulation finds tau that makes
 autocorrelation fall to 1/e.
    It stores these data in a dataframe and prints an html table of it
    :return: (void)
    """
    tmp_data_list = []
    T_val = [2.3]
    T_values = [round(T_val[i], 3) for i in range(len(T_val))]
    L_values = [10, 20, 30, 40, 50]
    print(f'Total number of samples: {len(L_values)*len(T_values)}')

    with open('correlation_table.txt', 'w') as file:
        for L in L_values:

            N = L**2
            s = get_spin_list(hot_start, N)
            neighbours_dictionary = get_neighbours_index(L)

            for T in T_values:

                start = time.process_time()
                b = 1 / T  # Constant: 1 / temperature
                dE_4 = exp(-4 * b)  # Probability for energy change of +4
                dE_8 = exp(-8 * b)  # Probability for energy change of +8
                target_tau = get_target_tau(s, L, dE_4, dE_8,
neighbours_dictionary)
                # tmp_data_list.append([N, T, target_tau])
                time_for_sample = time.process_time() - start
                print(f'[L = {L}, T = {T}, tau = {target_tau}] --> Time for
sample: {time_for_sample/60:.1f} minutes')
```

```
                file.write(f'{L},{T},{target_tau}\n')

def function_to_fit(L_array, z, a, b):
    """
    :param L_array: (nparray) Contains the L values tried
    :param z: (float) z exponent in critical slowing down equation
    :param a: (float) constant of equation
    :param b: (float) constant of equation
    :return: (nparray) The function to fit by finding z, a, b
    """
    return a*(L_array**z)+b

def critical_slowing_down_fit():
    """
    Tries to fit the data of tau vs L to find the z exponent in the critical slowing down relationship.
    It also makes a plot of the data and the fit and prints z
    :return: (void)
    """
    with open('correlation_table.txt', 'r') as file:

        L_list = []
        T_list = []
        tau_list = []

        line = file.readline()
        while line:
            row = line.strip().split(',')
            L_list.append(int(row[0]))
            T_list.append(float(row[1]))
            tau_list.append(float(row[2]))
            line = file.readline()

        L_list_df = pd.DataFrame(L_list, columns = ['L'])
        T_list_df = pd.DataFrame(T_list, columns = ['Temperature'])
        tau_list_df = pd.DataFrame(tau_list, columns = ['tau'])

        df = pd.concat([L_list_df, T_list_df, tau_list_df], axis = 1)
        df.to_html('L_T_tau2.html')
        print(df)

        tau_array = np.array(tau_list)
        L_array = np.array(L_list)

        p0 = [2.1, 0.2, 1]
        param_fit, cov = curve_fit(function_to_fit, L_array, tau_array, p0 = p0)
        print(f'z was found to be: {param_fit[0]}, with standard deviation: {np.sqrt(np.diag(cov)[0])}')
        z_fit, a_fit, b_fit = param_fit[0], param_fit[1], param_fit[2]
        tau_fit = a_fit*(L_array**z_fit) + b_fit

        plt.scatter(L_array, tau_array, marker = '+', color = 'r', s = 100)
        plt.plot(L_array, tau_fit, color = 'k')
        plt.legend([f'Fit: z = {z_fit:.3f}', 'Simulation'])
        plt.title('Critical slowing down $\u03C4_e$ vs L\nFit: $\u03C4_e$$\sim$$L^{z}$')
        plt.xlabel('L')
        plt.ylabel('$\u03C4_e$')
        plt.grid(True)
        # plt.savefig('critical_slowing_down.pdf', bbox = 'tight')
        plt.show()

critical_slowing_down_fit()
```

Code file: domains.py

```
from random import uniform, randint
```

```
import random
from math import exp
import matplotlib.pyplot as plt
import matplotlib.animation as anim
import numpy as np
import time
from collections import Counter
from queue import Queue

# Global variables
J = 1 # Exchange energy
H = 0
mu = 1 # Magnetic moment
number_of_neighbours = 4
random_seed = 13 # Fix random seed in spin initialization for reproducability
hot_start = True # Initialize with hot start or not
nsweeps = 10000 # Number of sweeps

def get_spin_list(is_hot, N):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make cold start
    :param N: (int) N = L x L, total number of sites
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(N):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N
    return s_local

def get_neighbours_index(N):
    """
    :param N: (int) N = L x L, total number of sites
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {}
    L = int(N**(1/2))
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
            bottom = i + L - N
```

```
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

def get_energy_difference(index_to_flip, s, neighbours_dictionary):
    """
    :param index_to_flip: (int) the site index to consider flipping its spin
    :param s: (list) spin list of the lattice
    :param neighbours_dictionary: (dict) holds indexes for each site's
     neighbours
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours + 2*s[index_to_flip]*
     mu*H
    return total_change

def metropolis(N, s, neighbours_dictionary, b):
    """
    The metropolis algorithm as a markov chain monte carlo simulation
     algorithm that modifies the spin state of the
    lattice and gives a new state by choosing N (= L x L) sites at random and
      checking through the energy if it will
    flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
     spin will be flipped and the numbers 4, 8
    represent the corresponding change in energy so that we don't calculate
     many times an exponential term
    :param b: (float) 1/Temperature
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (void) does not return anything but changes the state of s
    """
    for i in range(N):
        site_index = randint(0, N-1)
        dE = get_energy_difference(site_index, s, neighbours_dictionary)
        val = exp(-dE*b)
        rdm_num = uniform(0, 1)
        if dE <= 0:
            s[site_index] *= -1
        elif val > rdm_num:
            s[site_index] *= -1
        else:
            continue

def get_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Total magnetisation
    """
    magnetisation_total = 0
    for i in range(N):
        magnetisation_total += s[i]
    return magnetisation_total

def get_energy(N, s, neighbours_dictionary):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (float) Total energy through the Hamiltonian
    """
```

```
    sum1 = 0
    sum2 = 0
    for i in range(N):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation(N, s)
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy

def get_average_energy(N, s, neighbours_dictionary):
    return get_energy(N, s, neighbours_dictionary)/(2*N)

def get_average_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Magnetisation per site
    """
    return get_magnetisation(N, s)/N

def spin_configuration_2D(s, L):

    x, y = 0, L-1
    s_2D = [[0 for i in range(L)] for j in range(L)]
    for idx in range(len(s)):

        if (idx + 1) % L == 0:
            s_2D[x][y] = s[idx]
            x = 0
            y -= 1
        else:
            s_2D[x][y] = s[idx]
            x += 1
    return s_2D

def draw_lattice(s, L, T, avg_clust_size):

    s_2D = []

    for i in range(L):
        s_2D.append(s[i*L:(i+1)*L])

    s_2D_array = np.array(s_2D)
    plt.imshow(s_2D_array, cmap = 'binary')
    plt.title(f'L = {L}, T = {T:.2f}, Average cluster size = {avg_clust_size
    :.2f}')
    plt.show()

def bfs(s, index_to_start, neighbours_dictionary):
    """
    :param s: (list) lattice spin list
    :param index_to_start: (int) index to start calculating a cluster from
    :param neighbours_dictionary: (dictionary) holds indices for the site's
    neighbours
    :return: (int, list) cluster size, indices of sites that are now saved in
     a calculated cluster
    """
    cluster_type = s[index_to_start]
    seen = [False] * len(s)
    to_explore = Queue()
    to_explore.put(index_to_start)
    seen[index_to_start] = True
    cluster_size = 1
    cluster_indices = [index_to_start]

    while not to_explore.empty():
        idx = to_explore.get()
```

```
193         for neighbour_index in neighbours_dictionary[idx]:
194             if not seen[neighbour_index] and s[neighbour_index] ==
    cluster_type:
195                 to_explore.put(neighbour_index)
196                 seen[neighbour_index] = True
197                 cluster_size += 1
198                 cluster_indices.append(neighbour_index)
199 
200     return cluster_size, cluster_indices
201 
202 def get_average_cluster_size(s, neighbours_dictionary):
203 
204     cluster_size, cluster_indices = bfs(s, 0, neighbours_dictionary)
205 
206     cluster_size_list = [cluster_size]
207 
208     for index_to_start in range(len(s)):
209         index_to_start += 1
210         if index_to_start > len(s)-1:
211             break
212         elif index_to_start in cluster_indices:
213             continue
214         else:
215             cluster_size, cluster_indices_tmp = bfs(s, index_to_start,
    neighbours_dictionary)
216             cluster_size_list.append(cluster_size)
217             cluster_indices.extend(cluster_indices_tmp)
218 
219     final_cluster_size_list = []
220     maximum_value = max(cluster_size_list)
221 
222     for i in range(len(cluster_size_list)):
223 
224         if cluster_size_list[i] > maximum_value/2:
225             final_cluster_size_list.append(cluster_size_list[i])
226 
227     return np.average(final_cluster_size_list)
228 
229 def simulation():
230     """
231     The simulation function gathers data for average magnetisation vs
    temperature for different L dimensions of the
232     lattice. It stores the data in a pandas dataframe and makes a plot of all
     L in the same average magnetisation vs
233     temperature graph
234     :return: (void)
235     """
236     # Temperature values in Kelvin
237     T_extra = list(np.linspace(2.2, 2.5, 11))
238     T_values = list(np.linspace(2, 2.8, 31))
239     T_values.extend(T_extra)
240     T_values.sort()
241     # Dimension values
242     L_values = [12, 24, 32]
243     # Independent sampling
244     thermalisation_sweeps = 5000
245     sample_every = 50
246 
247     with open('domains.txt', 'w') as file:
248 
249         for L in L_values:
250 
251             N = L**2
252             s = get_spin_list(hot_start, N)
253             neighbours_dictionary = get_neighbours_index(N)
254 
255             for T in T_values:
256 
```

```
                b = 1/T  # Constant: 1 / temperature
                cluster_size_list = []
                start = time.process_time()

                for i in range(nsweeps):
                    metropolis(N, s, neighbours_dictionary, b)
                    if i < thermalisation_sweeps:
                        continue
                    elif i % sample_every == 0:
                        cluster_size_list.append(get_average_cluster_size(s,
    neighbours_dictionary))

                cluster_std = np.std(cluster_size_list)
                cluster_avg = np.average(cluster_size_list)

                file.write(f'{L},{T},{cluster_avg},{cluster_std}\n')
                time_for_sample = time.process_time() - start
                print(f'L = {L}, T = {T:.2f}, Cluster size = {cluster_avg:.2f
    }, Std = {cluster_std:.2f} --> Time for sample = {time_for_sample:.2f}
    seconds')

def domains_plotter():

    with open('domains.txt', 'r') as file:

        L_list = []
        T_list = []
        cluster_values = []
        cluster_error = []

        line = file.readline()
        while line:
            row = line.strip().split(',')
            L_list.append(int(row[0]))
            T_list.append(float(row[1]))
            cluster_values.append(float(row[2]))
            cluster_error.append(float(row[3]))
            line = file.readline()

        L_values = list(Counter(L_list).keys())
        T_values = list(Counter(T_list).keys())
        T_len = len(T_values)
        L_str_values = ['$T_c$']
        L_str_values.extend(['L = ' + str(L_values[i]) for i in range(len(
    L_values))])

        for i in range(len(L_values)-1):
            i += 1
            plt.errorbar(T_values, cluster_values[i*T_len:(i+1)*T_len], yerr
    = cluster_error[i*T_len:(i+1)*T_len],
                         marker = '+', label = f'L = {L_values[i]}',
    errorevery = 5, ecolor = 'r', capsize = 2)

        plt.axvline(x=2 / np.log(1 + np.sqrt(2)), color='k', linestyle='--',
    label='$T_c$')
        plt.legend()
        plt.xlabel('T (J/$k_B$)')
        plt.ylabel('Average Cluster size')
        plt.title('Average Cluster size vs Temperature')
        plt.grid()
        plt.savefig('domains1.pdf', bbox = 'tight')
        plt.show()

#simulation()
domains_plotter()
```

Code file: heat_capacity.py

```
from random import uniform, randint
import random
from math import exp
import numpy as np
from bootstrap import bootstrap
from heat_capacity_plotter import heat_capacity_plotter
import time
import pandas as pd

# Global variables
J = 1 # Exchange energy
H = 0 # Applied magnetic field strength
mu = 1 # Magnetic moment
M = 0 # Total magnetisation
random_seed = 13
number_of_neighbours = 4
hot_start = False # Initialize with hot start
nsweeps = 60000 # Number of sweeps

def get_spin_list(is_hot, N):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make cold start
    :param N: (int) N = L x L, total number of sites
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(N):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N
    return s_local

def get_neighbours_index(N):
    """
    :param N: (int) N = L x L, total number of sites
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {}
    L = int(N**(1/2))
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
```

```
            bottom = i + L - N
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

def get_energy_difference(index_to_flip, s, neighbours_dictionary):
    """
    :param index_to_flip: (int) the site index to consider flipping its spin
    :param s: (list) spin list of the lattice
    :param neighbours_dictionary: (dict) holds indexes for each site's
     neighbours
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours
    return total_change

def metropolis(dE_4, dE_8, N, s, neighbours_dictionary):
    """
    The metropolis algorithm as a markov chain monte carlo simulation
     algorithm that modifies the spin state of the
    lattice and gives a new state by choosing N (= L x L) sites at random and
      checking through the energy if it will
    flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
     spin will be flipped and the numbers 4, 8
    represent the corresponding change in energy so that we don't calculate
     many times an exponential term
    :param dE_4: (float) Probability for energy change of +4
    :param dE_8: (float) Probability for energy change of +8
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (void) does not return anything but changes the state of s
    """
    for i in range(N):
        site_index = randint(0, N-1)
        dE = get_energy_difference(site_index, s, neighbours_dictionary)
        rdm_num = uniform(0, 1)
        if dE <= 0:
            s[site_index] *= -1
        elif dE == 4:
            if dE_4 > rdm_num:
                s[site_index] *= -1
            else:
                continue
        elif dE == 8:
            if dE_8 > rdm_num:
                s[site_index] *= -1
            else:
                continue

def get_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Total magnetisation
    """
    magnetisation_total = 0
    for i in range(N):
        magnetisation_total += s[i]
    return magnetisation_total

def get_average_energy(N, s, neighbours_dictionary):
```

```
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (float) Total energy through the Hamiltonian
    """
    sum1 = 0
    sum2 = 0
    for i in range(N):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation(N, s)
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy/2*N

def get_energy(N, s, neighbours_dictionary):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (float) Total energy through the Hamiltonian
    """
    sum1 = 0
    sum2 = 0
    for i in range(N):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation(N, s)
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy

def get_average_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Magnetisation per site
    """
    return abs(get_magnetisation(N, s))/N

def get_autocovariance(M_list, tau):
    """
    :param M_list: (list) holding average magnetisation for each sweep
     neglected pre-thermalised samples
    :param tau: (int) time lag which is the input to the autocovariance
     formula
    :return: (float) autocovariance for the time lag tau
    """
    mean = np.average(M_list)
    autocovariance_list = [(M_list[t] - mean)*(M_list[t+tau] - mean) for t in
      range(len(M_list) - tau)]
    return  np.average(autocovariance_list)

def get_autocorrelation(M_list, tau):
    """
    :param M_list: (list) holding average magnetisation for each sweep
     neglected pre-thermalised samples
    :param tau: (int) time lag which is the input to the autocovariance
     formula
    :return: (float) autocorrelation for the time lag tau
    """
    A_0 = get_autocovariance(M_list, 0)
    A_tau = get_autocovariance(M_list, tau)
    return  A_tau/A_0

def get_target_value_index(autocorrelation_list, target_value):
    """
```

```
    :param autocorrelation_list: (list) autocorrelation for different tau values
    :param target_value: (float) autocorrelation initial value * 1/e
    :return: (int) index for target tau
    """
    target_index = -1
    for i in range(len(autocorrelation_list)):
        if autocorrelation_list[i] < target_value:
            target_index = i
            break
        else:
            continue
    return target_index

def get_target_tau(avg_mag_list):
    """
    :param avg_mag_list: (list) holds total magnetisation for each sweep state
    :return: (int) tau that makes autocorrelation fall to 1/e
    """
    autocorrelation_list = []
    tau_list = np.arange(0, 50, 1)

    for tau in tau_list:
        autocorrelation_list.append(get_autocorrelation(avg_mag_list, tau))

    target_value = 1/np.e
    index = get_target_value_index(autocorrelation_list, target_value)
    target_tau = tau_list[index]

    return target_tau

def simulation():
    """
    Generates data for C vs T for different values of L along with standard deviation for C using bootstrap.
    The data are written in a txt file and plotted using another file called heat_capacity_plotter.py. This
    function also finds the Tc by taking the T input that gives maximum C and  prints a table of L vs Tc
    :return: (void)
    """
    T_val = np.linspace(2, 3, 100)
    T_values = [round(T_val[i], 3) for i in range(len(T_val))]
    L_values = [64]
    print(f'Total samples to calculate: {len(T_values)*len(L_values)}')
    n_bins = 100
    Tc_list = []
    thermalisation_sweeps = 10000
    sample_every = 50

    with open('heat_capacity_data1.txt', 'w') as file:
        for L in L_values:

            C_list, sigma_C_list = [], []
            N = L ** 2
            s = get_spin_list(hot_start, N)
            neighbours_dictionary = get_neighbours_index(N)

            for T in T_values:

                start = time.process_time()
                avg_mag_list = []
                energy_list = []
                b = 1 / T  # Constant: 1 / temperature
                dE_4 = exp(-4 * b)  # Probability for energy change of +4
                dE_8 = exp(-8 * b)  # Probability for energy change of +8

```

```
                for sweep in range(nsweeps):

                    metropolis(dE_4, dE_8, N, s, neighbours_dictionary)

                    if sweep < thermalisation_sweeps:
                        continue
                    else:
                        if sweep % sample_every == 0:
                            avg_mag_list.append(get_average_magnetisation(N, s))
                            energy_list.append(get_energy(N, s, neighbours_dictionary))

                target_tau = get_target_tau(avg_mag_list)
                C, sigma_C = bootstrap(energy_list, n_bins, T, target_tau)
                C_list.append(C)
                sigma_C_list.append(sigma_C)
                time_for_sample = time.process_time() - start

                file.write(f'{L},{T},{C},{sigma_C}\n')
                print(f'[L = {L}, T = {T}, C = {C:.2f}, sigma = {sigma_C:.2f}, tau = {target_tau}]', f'--> Time for sample: {time_for_sample/60:.1f} minutes')

            index_for_Tc = C_list.index(max(C_list))
            Tc_list.append(T_values[index_for_Tc])

        L_df = pd.DataFrame(L_values, columns = ['L'])
        Tc_df = pd.DataFrame(Tc_list, columns = ['Tc'])
        Tc_data = pd.concat([L_df, Tc_df], axis = 1)
        print(Tc_data)
        Tc_data.to_html('Tc_vs_L_table1.html')

simulation()
heat_capacity_plotter()
```

Code file: heat_capacity_plotter.py

```
import matplotlib.pyplot as plt
from collections import Counter
import numpy as np
from scipy.optimize import curve_fit
import pandas as pd

def heat_capacity_plotter():
    """
    Plots C/N vs T for different values of L
    :return: (void)
    """
    with open('heat_capacity_data1.txt', 'r') as file:

        onsager_Tc = 2/np.log(1+np.sqrt(2))

        L_list = []
        T_list = []
        C_list = []
        sigma_C_list = []

        line = file.readline()
        while line:
            row = line.strip().split(',')
            L_list.append(int(row[0]))
            T_list.append(float(row[1]))
            C_list.append(float(row[2]))
            sigma_C_list.append(float(row[3]))
            line = file.readline()

        L_values = list(Counter(L_list).keys())
```

```
        L_length = len(L_values)
        T_length = len(list(Counter(T_list).keys()))

        L_str_values = ['L = ' + str(L_values[i]) for i in range(len(L_values))]
        L_str_values.insert(0, 'Onsager\'s $T_c$')

        C_array = np.array(C_list)
        sigma_C_array = np.array(sigma_C_list)
        N_values_array = np.array(L_values)**2

        colour_list = ['k', 'b', 'g', 'y', 'm', 'c', 'r', 'navy', 'lightcoral', 'lime']

        for i in range(L_length):
            plt.errorbar(T_list[i*T_length:(i+1)*T_length], C_array[i*T_length:(i+1)*T_length]/N_values_array[i],
                         xerr = 0, yerr = sigma_C_array[i*T_length:(i+1)*T_length]/N_values_array[i], ls = '',
                         marker = '+', ecolor = 'r', capsize = 2, color = colour_list[i], elinewidth = 0.7, errorevery = 30)


        plt.axvline(x=onsager_Tc, color='k', linestyle='--')
        # plt.yscale('log')
        plt.legend(L_str_values)
        plt.title('Specific heat capacity per site vs temperature')
        plt.xlabel('T (J/$k_B$)')
        plt.ylabel('C/N')
        # plt.savefig('heat_cap_vs_tmp2.pdf', bbox = 'tight')
        plt.show()

        tc_fitter(L_list, T_list, C_list, sigma_C_list)

def gaussian(x, mu, sigma, amp):
    return amp*np.exp(-((x-mu)/sigma)**2)

def tc_fitter(L_values, T_values, C_values, sigma_C_values):

    # L_values = [4, 8, 10, 12, 24, 32, 48]
    # tc_values = [2.394, 2.354, 2.364, 2.384, 2.293, 2.293, 2.273]

    L_val = list(Counter(L_values).keys())
    L_len = len(L_val)
    T_len = len(list(Counter(T_values).keys()))

    tc_list = []
    tc_error = []

    with open('finite_scaling_data.txt', 'w') as file:
        for i in range(L_len):

            T_tmp = T_values[i*T_len:(i+1)*T_len]
            C_tmp = C_values[i*T_len:(i+1)*T_len]
            sigma_C_tmp = sigma_C_values[i*T_len:(i+1)*T_len]
            index_of_max = C_tmp.index(max(C_tmp))
            T = T_tmp[index_of_max - 20:index_of_max + 20]
            C = C_tmp[index_of_max - 20:index_of_max + 20]
            C_sigma = sigma_C_tmp[index_of_max - 20:index_of_max + 20]
            parameters, pcov = curve_fit(gaussian, T, C, sigma = C_sigma, absolute_sigma = True)
            mu, sigma, amp, sigma_mu = parameters[0], parameters[1], parameters[2], np.sqrt(np.diag(pcov)[0])
            tc_list.append(mu)
            tc_error.append(sigma_mu)
            file.write(f'{L_val[i]**2},{mu},{sigma_mu}\n')

    L_df = pd.DataFrame(L_val, columns = ['L'])
```

```
    tc_df = pd.DataFrame(tc_list, columns = ['Tc'])
    tc_error_df = pd.DataFrame(tc_error, columns = ['sigma_Tc'])
    df = pd.concat([L_df, tc_df, tc_error_df], axis = 1)
    # df.to_html('L_Tc_error.html')
    print(df)


def func(L, tc_at_inf, a, nu):
    return tc_at_inf + a*L**(-1/nu)

def finite_scaling_fitter():

    with open('finite_scaling_data.txt', 'r') as file:

        L_list = []
        tc_list = []
        tc_error = []

        line = file.readline()
        while line:
            row = line.strip().split(',')
            L_list.append(float(row[0]))
            tc_list.append(float(row[1]))
            tc_error.append(float(row[2]))
            line = file.readline()

        p0 = [2.26, 1.0, 2.0]

        parameters, covariance_matrix = curve_fit(func, L_list, tc_list,
                                                  sigma = tc_error,
    absolute_sigma = True, p0 = p0)
        tc_at_inf, a, nu = parameters[0], parameters[1], parameters[2]
        tc_at_inf_error, a_error, nu_error = np.sqrt(np.diag(
    covariance_matrix)[0]), \
                                             np.sqrt(np.diag(
    covariance_matrix)[1]),\
                                             np.sqrt(np.diag(
    covariance_matrix)[2])

        print(f'\nTc at infinity: {tc_at_inf:.3f}, sigma: {tc_at_inf_error:.3
    f}\na: {a:.3f}, '
              f'sigma: {a_error:.3f}\nnu: {nu:.3f}, sigma: {nu_error:.3f}')

        plt.plot(L_list, tc_list, color = 'k', linestyle = '--')
        plt.plot(L_list, tc_at_inf + a*np.array(L_list)**(-1/nu), color = 'r'
    )
        plt.title('Finite-size scaling: $T_c$ vs N\nFit: $T_c(N)$ = $T_c(\
    u221e) + \u03B1(N)^{-\\frac{1}{\u03BD}}$')
        plt.ylabel('$T_c(N)$')
        plt.xlabel('N')
        plt.legend(['Simulation', 'Fit'])
        plt.savefig('finite_size_scaling.pdf', bbox = 'tight')
        plt.show()

heat_capacity_plotter()
finite_scaling_fitter()
```

Code file: hysteresis.py

```
from random import uniform, randint
import random
from math import exp
import matplotlib.pyplot as plt
import numpy as np
import time
from collections import Counter
import turtle

```

```
# Global variables
J = 1 # Exchange energy
mu = 1 # Magnetic moment
number_of_neighbours = 4
random_seed = 13 # Fix random seed in spin initialization for reproducability
hot_start = False # Initialize with hot start or not
nsweeps = 8000 # Number of sweeps

def get_spin_list(is_hot, N):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make cold start
    :param N: (int) N = L x L, total number of sites
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(N):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N
    return s_local

def get_neighbours_index(N):
    """
    :param N: (int) N = L x L, total number of sites
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {}
    L = int(N**(1/2))
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
            bottom = i + L - N
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

def get_energy_difference(index_to_flip, s, neighbours_dictionary, H):
    """
    :param H: (float) Applied magnetic field
```

```
74      :param index_to_flip: (int) the site index to consider flipping its spin
75      :param s: (list) spin list of the lattice
76      :param neighbours_dictionary: (dict) holds indexes for each site's
        neighbours
77      :return: (float) Total energy change of changing that site's spin
78      """
79      sum_of_neighbours = 0
80      for neighbour_index in neighbours_dictionary[index_to_flip]:
81          sum_of_neighbours += s[neighbour_index]
82      total_change = 2*s[index_to_flip]*sum_of_neighbours + 2*s[index_to_flip]*
        mu*H
83      return total_change
84
85  def metropolis(N, s, neighbours_dictionary, H, b):
86      """
87      The metropolis algorithm as a markov chain monte carlo simulation
        algorithm that modifies the spin state of the
88      lattice and gives a new state by choosing N (= L x L) sites at random and
          checking through the energy if it will
89      flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
        spin will be flipped and the numbers 4, 8
90      represent the corresponding change in energy so that we don't calculate
        many times an exponential term
91      :param b: (float) 1/Temperature
92      :param H: (float) Applied magnetic field
93      :param N: (int) total number of sites
94      :param s: (list) spin list
95      :param neighbours_dictionary: (dict) holds indexes for neighbours
96      :return: (void) does not return anything but changes the state of s
97      """
98      for i in range(N):
99          site_index = randint(0, N-1)
100         dE = get_energy_difference(site_index, s, neighbours_dictionary, H)
101         val = exp(-dE*b)
102         rdm_num = uniform(0, 1)
103         if dE <= 0:
104             s[site_index] *= -1
105         elif val > rdm_num:
106             s[site_index] *= -1
107         else:
108             continue
109
110 def get_magnetisation(N, s):
111     """
112     :param N: (int) total number of sites
113     :param s: (list) spin list
114     :return: (float) Total magnetisation
115     """
116     magnetisation_total = 0
117     for i in range(N):
118         magnetisation_total += s[i]
119     return magnetisation_total
120
121 def get_energy(N, s, neighbours_dictionary, H):
122     """
123     :param H: (float) Applied magnetic field
124     :param N: (int) total number of sites
125     :param s: (list) spin list
126     :param neighbours_dictionary: (dict) holds indexes for neighbours
127     :return: (float) Total energy through the Hamiltonian
128     """
129     sum1 = 0
130     sum2 = 0
131     for i in range(N):
132         for j in range(number_of_neighbours):
133             sum1 += s[i]*s[neighbours_dictionary[i][j]]
134         if H != 0:
135             sum2 = get_magnetisation(N, s)
```

```
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy

def get_average_energy(N, s, neighbours_dictionary, H):
    return get_energy(N, s, neighbours_dictionary, H)/(2*N)

def get_average_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Magnetisation per site
    """
    return get_magnetisation(N, s)/N

def spin_configuration_2D(s, L):

    x, y = 0, L-1
    s_2D = [[0 for i in range(L)] for j in range(L)]
    for idx in range(len(s)):

        if (idx + 1) % L == 0:
            s_2D[x][y] = s[idx]
            x = 0
            y -= 1
        else:
            s_2D[x][y] = s[idx]
            x += 1
    return s_2D

def draw_lattice(s, L, T, avg_clust_size):

    s_2D = []

    for i in range(L):
        s_2D.append(s[i*L:(i+1)*L])

    s_2D_array = np.array(s_2D)
    plt.imshow(s_2D_array, cmap = 'binary')
    plt.title(f'L = {L}, T = {T:.2f}, Average cluster size = {avg_clust_size
    :.2f}')
    plt.show()

def simulation():
    """
    The simulation function gathers data for average magnetisation vs
     temperature for different L dimensions of the
    lattice. It stores the data in a pandas dataframe and makes a plot of all
     L in the same average magnetisation vs
    temperature graph
    :return: (void)
    """
    # Temperature values in Kelvin
    T_values = [2.4, 2.6, 3]
    # Dimension values
    L_values = [10]
    # Applied magnetic field values
    extra_H_neg = list(np.linspace(-0.05128, -0.05385, 5))
    H_val = list(np.linspace(0, 1.0, 21))
    H_val.extend(H_val[::-1])
    H_neg = list(np.linspace(0, -1.0, 21))
    H_neg.extend(extra_H_neg)
    H_neg.sort(reverse=True)
    H_neg.extend(H_neg[::-1])
    H_val.extend(H_neg)
    H_values = [round(H_val[i], 10) for i in range(len(H_val))]
    tmp_list = [H_values[0]]

    for i in range(len(H_values) - 1):
```

```
201
202         if H_values[i] != H_values[i + 1]:
203             tmp_list.append(H_values[i + 1])
204
205     H_values = tmp_list
206     # Independent sampling
207     thermalisation_sweeps = 3000
208     sample_every = 20
209
210     with open('hysteresis.txt', 'w') as file:
211
212         for L in L_values:
213
214             for T in T_values:
215
216                 N = L ** 2
217                 s = get_spin_list(hot_start, N)
218                 neighbours_dictionary = get_neighbours_index(N)
219                 b = 1/T  # Constant: 1 / temperature
220
221                 for H in H_values:
222
223                     start = time.process_time()
224                     average_magnetisation_list = []
225                     energy_per_site_list = []
226
227                     for i in range(nsweeps):
228
229                         metropolis(N, s, neighbours_dictionary, H, b)
230                         if i < thermalisation_sweeps:
231                             continue
232                         elif i % sample_every == 0:
233                             mag_per_site = get_average_magnetisation(N, s)
234                             average_magnetisation_list.append(mag_per_site)
235                             energy_per_site_list.append(get_average_energy(N,
     s, neighbours_dictionary, H))
236
237                     mean_average_magnetisation = np.average(
    average_magnetisation_list)
238                     mean_energy = np.average(energy_per_site_list)
239                     file.write(f'{L},{T},{H},{mean_average_magnetisation},{
    mean_energy}\n')
240                     time_for_sample = time.process_time() - start
241                     print(f'L = {L}, T = {T}, H = {H}, <m> = {
    mean_average_magnetisation:.5f}, <E> = {mean_energy:.2f} --> Time for
    sample = {time_for_sample:.2f} seconds')
242
243 def hysteresis_plotter():
244
245     magnetisation_per_site_list = []
246     energy_per_site = []
247     L_list = []
248     T_list = []
249     H_list = []
250
251     with open('hysteresis.txt', 'r') as file:
252
253         line = file.readline()
254         count = 0
255
256         while line:
257             row = line.strip().split(',')
258             L_list.append(int(row[0]))
259             T_list.append(float(row[1]))
260             magnetisation_per_site_list.append(float(row[3]))
261             energy_per_site.append(float(row[4]))
262             if count == 3:
263                 line = file.readline()
```

```
                continue
            else:
                H_list.append(float(row[2]))
                if float(row[2]) == 0:
                    count += 1
            line = file.readline()

        L_values = list(Counter(L_list).keys())
        T_values = list(Counter(T_list).keys())
        H_values = H_list
        H_len = len(H_values)
        T_str_values = ['T = ' + str(T_values[i]) for i in range(len(T_values
    ))]

        for i in range(len(T_values)):
            plt.scatter(H_values, magnetisation_per_site_list[i*H_len:(i+1)*
    H_len], marker = '+', s = None)
        plt.title(f'Hysteresis loop (L = {L_values[0]})\nMagnetisation per
    site vs applied field')
        plt.xlabel('H')
        plt.ylabel('<m>')
        plt.legend(T_str_values)
        plt.grid()
        # plt.savefig('hysteresis_mag.pdf', bbox = 'tight')
        plt.show()

        for i in range(len(T_values)):
            plt.scatter(H_values, energy_per_site[i * H_len:(i + 1) * H_len],
     marker = '+', s = None)
        plt.title(f'Energy per site vs applied field (L = {L_values[0]})')
        plt.xlabel('H')
        plt.ylabel('<E>')
        plt.grid()
        plt.legend(T_str_values)
        # plt.savefig('hysteresis_energy.pdf', bbox = 'tight')
        plt.show()

#simulation()
hysteresis_plotter()
```

Code file: nearest_neighbours.py

```
from random import uniform, randint
import random
from math import exp
from nearest_neighbours_plottter import nearest_neighbours_plotter
import numpy as np
import time

# Global variables
H = 0 # Applied magnetic field strength
mu = 1 # Magnetic moment
M = 0 # Total magnetisation
random_seed = 10 # Fix random seed in spin initialization for reproducability
hot_start = False # Initialize with hot start or not
nsweeps = 2000 # Number of sweeps

def get_spin_list(is_hot, N):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make
     cold start
    :param N: (int) N = L x L, total number of sites
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
```

```
        for i in range(N):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N
    return s_local

def get_neighbours_index(N):
    """
    :param N: (int) N = L x L, total number of sites
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {}
    L = int(N**(1/2))
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
            bottom = i + L - N
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

def get_nearest_neighbours_index(N):
    """
    :param N: (int) Total number of sites
    :return: (dictionary) Keys - sites indices, values - next nearest neighbours' indices
    """
    L = int(N ** (1 / 2)) # Lattice dimension
    nearest_neighbours_dictionary = {} # Dictionary to return

    for i in range(N):
        # Up-left neighbour
        if i == 0: # If the site is index 0
            upleft = N - 1
        elif i % L == 0: # If the site is in the first column
            upleft = i - 1
        elif i - L < 0: # If the site is in the first row
            upleft = i - L + N - 1
        else:
            upleft = i - L - 1
        # Up-right neighbour
        if i == L - 1: # If the site is at the top right corner
            upright = L * (L - 1)
```

```
        elif (i + 1) % L == 0: # If the site is in the last column
            upright = i - 2 * L + 1
        elif i - L < 0: # If the site is in the first row
            upright = i - L + N + 1
        else:
            upright = i - L + 1
        # Down-left neighbour
        if i == L * (L - 1): # If the site is at the bottom left corner
            downleft = L - 1
        elif i + L >= N: # If the site is in the last row
            downleft = i + L - N - 1
        elif i % L == 0: # If the site is in the first column
            downleft = i + 2 * L - 1
        else:
            downleft = i + L - 1
        # Down-right neighbour
        if i == N - 1: # If the site is at the down right corner
            downright = 0
        elif (i + 1) % L == 0: # If the site is in the last column
            downright = i + 1
        elif i + L >= N: # If the site is in the last row
            downright = i + L - N + 1
        else:
            downright = i + L + 1

        nearest_neighbours_dictionary[i] = [upleft, upright, downleft,
    downright]

    return nearest_neighbours_dictionary

def get_energy_difference(index_to_flip, s, neighbours_dictionary,
    nearest_neighbours_dictionary, J):
    """
    :param index_to_flip: (int) the site index to consider flipping its spin
    :param s: (list) spin list of the lattice
    :param neighbours_dictionary: (dict) holds indexes for each site's
    neighbours
    :param nearest_neighbours_dictionary: (dict) holds indexes for each site'
    s nearest neighbours
    :param J: (float) exchange energy
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    sum_of_nearest_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    for nearest_neighbour_index in nearest_neighbours_dictionary[
    index_to_flip]:
        sum_of_nearest_neighbours += s[nearest_neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours + 2*J*s[index_to_flip
    ]*sum_of_nearest_neighbours
    return total_change

def metropolis(N, s, neighbours_dictionary, nearest_neighbours_dictionary, J,
     b):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :param nearest_neighbours_dictionary: (dict) holds indexes for nearest
    neighbours
    :param J: (float) exchange energy for next-nearest neighbours only (J=1
    for neighbours)
    :param b: (float) 1/Temperature
    :return: (void) does not return anything but changes the state of s
    """
    for i in range(N):
        site_index = randint(0, N-1)
```

```
149         dE = get_energy_difference(site_index, s, neighbours_dictionary,
    nearest_neighbours_dictionary, J)
150         val = exp(-dE * b)
151         rdm_num = uniform(0, 1)
152         if dE <= 0:
153             s[site_index] *= -1
154         elif val > rdm_num:
155             s[site_index] *= -1
156         else:
157             continue
158 
159 def get_magnetisation(N, s):
160     """
161     :param N: (int) total number of sites
162     :param s: (list) spin list
163     :return: (float) Total magnetisation
164     """
165     magnetisation_total = 0
166     for i in range(N):
167         magnetisation_total += s[i]
168     return magnetisation_total
169 
170 def get_average_magnetisation(N, s):
171     """
172     :param N: (int) total number of sites
173     :param s: (list) spin list
174     :return: (float) Magnetisation per site
175     """
176     return abs(get_magnetisation(N, s))/N
177 
178 def simulation():
179 
180     with open('nearest_neighbours1.txt', 'w') as file:
181 
182         # Temperature values in Kelvin
183         T_values = np.linspace(1, 10, 51)
184         # Dimension values
185         L_values = [64]
186         # Exchange energy values
187         J_values = [1, 0.7, 0.5]
188         # Sampling
189         thermalisation_sweeps = 1000
190         sample_every = 50
191 
192         for L in L_values:
193 
194             N = L**2
195             s = get_spin_list(hot_start, N)
196             neighbours_dictionary = get_neighbours_index(N)
197             nearest_neighbours_dictionary = get_nearest_neighbours_index(N)
198 
199             for J in J_values:
200 
201                 for T in T_values:
202 
203                     start = time.process_time()
204                     b = 1 / T  # Constant: 1 / temperature
205                     tmp_magnetisation = []
206 
207                     for i in range(nsweeps):
208                         metropolis(N, s, neighbours_dictionary,
    nearest_neighbours_dictionary, J, b)
209                         if i < thermalisation_sweeps:
210                             continue
211                         elif i % sample_every == 0:
212                             tmp_magnetisation.append(
    get_average_magnetisation(N, s))
213 
```

```
                    time_for_sample = time.process_time() - start
                    file.write(f'{L},{J},{T},{np.average(tmp_magnetisation)}\n')
                    print(f'L = {L}, J = {J}, T = {T:.2f}, <m> = {np.average(tmp_magnetisation):.5f}, Time for sample = {time_for_sample:.2f} seconds')

simulation()
nearest_neighbours_plotter()
```

Code file: susceptibility.py

```
from random import uniform, randint
import random
from math import exp
import numpy as np
from bootstrap_susceptibility import bootstrap
from susceptibility_plotter import susceptibility_plotter
import time

# Global variables
J = 1 # Exchange energy
H = 0 # Applied magnetic field strength
mu = 1 # Magnetic moment
M = 0 # Total magnetisation
random_seed = 11
number_of_neighbours = 4
hot_start = True # Initialize with hot start
nsweeps = 10000 # Number of sweeps

def get_spin_list(is_hot, N):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make cold start
    :param N: (int) N = L x L, total number of sites
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(N):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N
    return s_local

def get_neighbours_index(N):
    """
    :param N: (int) N = L x L, total number of sites
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {}
    L = int(N**(1/2))
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
```

```
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
            bottom = i + L - N
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

def get_energy_difference(index_to_flip, s, neighbours_dictionary):
    """
    :param index_to_flip: (int) the site index to consider flipping its spin
    :param s: (list) spin list of the lattice
    :param neighbours_dictionary: (dict) holds indexes for each site's
     neighbours
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours
    return total_change

def metropolis(dE_4, dE_8, N, s, neighbours_dictionary):
    """
    The metropolis algorithm as a markov chain monte carlo simulation
     algorithm that modifies the spin state of the
    lattice and gives a new state by choosing N (= L x L) sites at random and
      checking through the energy if it will
    flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
     spin will be flipped and the numbers 4, 8
    represent the corresponding change in energy so that we don't calculate
     many times an exponential term
    :param dE_4: (float) Probability for energy change of +4
    :param dE_8: (float) Probability for energy change of +8
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (void) does not return anything but changes the state of s
    """
    for i in range(N):
        site_index = randint(0, N-1)
        dE = get_energy_difference(site_index, s, neighbours_dictionary)
        rdm_num = uniform(0, 1)
        if dE <= 0:
            s[site_index] *= -1
        elif dE == 4:
            if dE_4 > rdm_num:
                s[site_index] *= -1
            else:
                continue
        elif dE == 8:
            if dE_8 > rdm_num:
                s[site_index] *= -1
            else:
                continue

def get_magnetisation(N, s):
```

```
116     """
117     :param N: (int) total number of sites
118     :param s: (list) spin list
119     :return: (float) Total magnetisation
120     """
121     magnetisation_total = 0
122     for i in range(N):
123         magnetisation_total += s[i]
124     return magnetisation_total
125 
126 def get_average_energy(N, s, neighbours_dictionary):
127     """
128     :param N: (int) total number of sites
129     :param s: (list) spin list
130     :param neighbours_dictionary: (dict) holds indexes for neighbours
131     :return: (float) Total energy through the Hamiltonian
132     """
133     sum1 = 0
134     sum2 = 0
135     for i in range(N):
136         for j in range(number_of_neighbours):
137             sum1 += s[i]*s[neighbours_dictionary[i][j]]
138         if H != 0:
139             sum2 = get_magnetisation(N, s)
140     total_energy = (-J*sum1 - mu*H*sum2)/2
141     return total_energy/2*N
142 
143 def get_energy(N, s, neighbours_dictionary):
144     """
145     :param N: (int) total number of sites
146     :param s: (list) spin list
147     :param neighbours_dictionary: (dict) holds indexes for neighbours
148     :return: (float) Total energy through the Hamiltonian
149     """
150     sum1 = 0
151     sum2 = 0
152     for i in range(N):
153         for j in range(number_of_neighbours):
154             sum1 += s[i]*s[neighbours_dictionary[i][j]]
155         if H != 0:
156             sum2 = get_magnetisation(N, s)
157     total_energy = (-J*sum1 - mu*H*sum2)/2
158     return total_energy
159 
160 def get_average_magnetisation(N, s):
161     """
162     :param N: (int) total number of sites
163     :param s: (list) spin list
164     :return: (float) Magnetisation per site
165     """
166     return abs(get_magnetisation(N, s))/N
167 
168 def get_autocovariance(M_list, tau):
169     """
170     :param M_list: (list) holding average magnetisation for each sweep
      neglected pre-thermalised samples
171     :param tau: (int) time lag which is the input to the autocovariance
      formula
172     :return: (float) autocovariance for the time lag tau
173     """
174     mean = np.average(M_list)
175     autocovariance_list = [(M_list[t] - mean)*(M_list[t+tau] - mean) for t in
       range(len(M_list) - tau)]
176     return  np.average(autocovariance_list)
177 
178 def get_autocorrelation(M_list, tau):
179     """
180     :param M_list: (list) holding average magnetisation for each sweep
```

```
    neglected pre-thermalised samples
    :param tau: (int) time lag which is the input to the autocovariance formula
    :return: (float) autocorrelation for the time lag tau
    """
    A_0 = get_autocovariance(M_list, 0)
    A_tau = get_autocovariance(M_list, tau)
    return  A_tau/A_0

def get_target_value_index(autocorrelation_list, target_value):
    """
    :param autocorrelation_list: (list) autocorrelation for different tau values
    :param target_value: (float) autocorrelation initial value * 1/e
    :return: (int) index for target tau
    """
    target_index = -1
    for i in range(len(autocorrelation_list)):
        if autocorrelation_list[i] < target_value:
            target_index = i
            break
        else:
            continue
    return target_index

def get_target_tau(avg_mag_list):
    """
    :param avg_mag_list: (list) holds total magnetisation for each sweep state
    :return: (int) tau that makes autocorrelation fall to 1/e
    """
    autocorrelation_list = []
    tau_list = np.arange(0, 50, 1)

    for tau in tau_list:
        autocorrelation_list.append(get_autocorrelation(avg_mag_list, tau))

    target_value = 1/np.e
    index = get_target_value_index(autocorrelation_list, target_value)
    target_tau = tau_list[index]

    return target_tau

def simulation():
    """
    Generates data for chi vs T for different values of L along with standard deviation for chi using
    bootstrap_susceptibility. The data are written in a txt file and plotted using another file called
    susceptibility_plotter.py.
    :return: (void)
    """
    T_val = np.linspace(1, 4, 51)
    T_values = [round(T_val[i], 3) for i in range(len(T_val))]
    L_values = [16, 24]
    print(f'Total samples to calculate: {len(T_values)*len(L_values)}')
    n_bins = 100
    thermalisation_sweeps = 5000
    sample_every = 80

    with open('susceptibility.txt', 'w') as file:
        for L in L_values:

            chi_list, sigma_chi_list = [], []
            N = L ** 2
            s = get_spin_list(hot_start, N)
            neighbours_dictionary = get_neighbours_index(N)

```

```
            for T in T_values:

                start = time.process_time()
                mag_list = []
                b = 1 / T  # Constant: 1 / temperature
                dE_4 = exp(-4 * b)  # Probability for energy change of +4
                dE_8 = exp(-8 * b)  # Probability for energy change of +8

                for sweep in range(nsweeps):

                    metropolis(dE_4, dE_8, N, s, neighbours_dictionary)

                    if sweep < thermalisation_sweeps:
                        continue
                    else:
                        if sweep % sample_every == 0:
                            mag_list.append(get_average_magnetisation(N, s))

                target_tau = get_target_tau(mag_list)
                chi, sigma_chi = bootstrap(mag_list, n_bins, T, target_tau)
                chi_list.append(chi)
                sigma_chi_list.append(sigma_chi)
                time_for_sample = time.process_time() - start

                file.write(f'{L},{T},{chi},{sigma_chi}\n')
                print(f'[L = {L}, T = {T}, chi = {chi:.8f}, sigma = {
    sigma_chi:.8f}, tau = {target_tau}]', f'--> Time for sample: {
    time_for_sample:.1f} seconds')

simulation()
susceptibility_plotter()
```

Code file: 3d_lattice.py

```
from random import uniform, randint
import random
from math import exp
from lattice_3d_plotter import lattice_3d_plotter
import numpy as np
import time

# Global variables
H = 0 # Applied magnetic field strength
mu = 1 # Magnetic moment
M = 0 # Total magnetisation
J = 1 # Exchange energy
number_of_neighbours = 6 # Number of neighbours in 3D
random_seed = 10 # Fix random seed in spin initialization for reproducability
hot_start = False # Initialize with hot start or not
nsweeps = 4000 # Number of sweeps

def get_spin_list(is_hot, N):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make
    cold start
    :param N: (int) N = L x L, total number of sites
    :return: (list) containing +/- 1 representing the spins on the lattice
    """
    random.seed(random_seed)
    L = int(N**(1/2))
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(L**3):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
```

```
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * (L**3)
    return s_local

def get_neighbours_index(N):
    """
    :param N: N = L x L
    :return: (dictionary) key is site value is list of neighbour indices
    """
    L = int(N**(1/2))
    neighbours_dictionary = {}
    lattice_indices = [[0 for k in range(N)] for j in range(L)]

    for i in range(L):
        for j in range(N):
            lattice_indices[i][j] = j + i*N
    for i in range(L):
        for j in range(N):

            idx = j + i*N

            if idx % L == 0:
                left = lattice_indices[i][j + L - 1]
            else:
                left = lattice_indices[i][j - 1]
            if (idx + 1) % L == 0:
                right = lattice_indices[i][j - L + 1]
            else:
                right = lattice_indices[i][j + 1]
            if idx - L < 0:
                top = lattice_indices[i][j - L + N]
            else:
                top = lattice_indices[i][j - L]
            if idx + L >= N:
                bottom = lattice_indices[i][j + L - N]
            else:
                bottom = lattice_indices[i][j + L]
            if i == 0:
                up = lattice_indices[L-1][j]
            else:
                up = lattice_indices[i-1][j]
            if i == L-1:
                down = lattice_indices[0][j]
            else:
                down = lattice_indices[i+1][j]

            neighbours_dictionary[idx] = [left, right, top, bottom, up, down]
    return neighbours_dictionary

def get_energy_difference(index_to_flip, s, neighbours_dictionary):
    """
    :param index_to_flip: (int) the site index to consider flipping its spin
    :param s: (list) spin list of the lattice
    :param neighbours_dictionary: (dict) holds indexes for each site's
     neighbours
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours
    return total_change

def metropolis(N, s, neighbours_dictionary, b):
    """
    :param N: (int) total number of sites
```

```
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :param b: (float) 1/Temperature
    :return: (void) does not return anything but changes the state of s
    """
    L = int(N**(1/2))
    for i in range(L**3):
        site_index = randint(0, (L**3)-1)
        dE = get_energy_difference(site_index, s, neighbours_dictionary)
        val = exp(-dE * b)
        rdm_num = uniform(0, 1)
        if dE <= 0:
            s[site_index] *= -1
        elif val > rdm_num:
            s[site_index] *= -1
        else:
            continue

def get_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Total magnetisation
    """
    magnetisation_total = 0
    L = int(N**(1/2))
    for i in range(L**3):
        magnetisation_total += s[i]
    return magnetisation_total

def get_average_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Magnetisation per site
    """
    L = int(N**(1/2))
    return abs(get_magnetisation(N, s))/(L**3)

def get_energy(N, s, neighbours_dictionary):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (float) Total energy through the Hamiltonian
    """
    sum1 = 0
    sum2 = 0
    L = int(N**(1/2))
    for i in range(L**3):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation(N, s)
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy

def get_average_energy(N, s, neighbours_dictionary):
    L = int(N**(1/2))
    return get_energy(N, s, neighbours_dictionary)/(3*(L**3))

def simulation():

    with open('lattice_3d.txt', 'w') as file:

        # Temperature values in Kelvin
        T_values = np.linspace(2, 6, 51)
        # Dimension values
```

```
        L_values = [12, 14]
        # Sampling
        thermalisation_sweeps = 2000
        sample_every = 50

        for L in L_values:

            N = L**2
            s = get_spin_list(hot_start, N)
            neighbours_dictionary = get_neighbours_index(N)

            for T in T_values:

                start = time.process_time()
                b = 1 / T  # Constant: 1 / temperature
                tmp_magnetisation = []
                tmp_energy = []

                for i in range(nsweeps):
                    metropolis(N, s, neighbours_dictionary, b)
                    if i < thermalisation_sweeps:
                        continue
                    elif i % sample_every == 0:
                        tmp_magnetisation.append(get_average_magnetisation(N, s))
                        tmp_energy.append(get_average_energy(N, s, neighbours_dictionary))

                time_for_sample = time.process_time() - start
                file.write(f'{L},{T},{np.average(tmp_magnetisation)},{np.average(tmp_energy)}\n')
                print(f'L = {L}, T = {T:.2f}, <m> = {np.average(tmp_magnetisation):.5f}, <E> = {np.average(tmp_energy):.5f}, Time for sample = {time_for_sample:.2f} seconds')

simulation()
lattice_3d_plotter()
```

Code file: deep_ising_temp.py

```
from random import uniform, randint
import random
from math import exp
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.models import load_model
from keras.layers import Conv2D
from hyperopt import hp
from keras.utils import plot_model

# Global variables
J = 1 # Exchange energy
L = 10 # Lattice dimension
H = 0 # Applied magnetic field strength
mu = 1 # Magnetic moment
M = 0 # Total magnetisation
number_of_neighbours = 4
hot_start = True # Initialize with hot start or not
nsweeps = 500 # Number of sweeps

def get_spin_list(is_hot, N):
    """
    :param is_hot: (bool) If is_hot is true make hot start otherwise make cold start
    :param N: (int) N = L x L, total number of sites
    :return: (list) containing +/- 1 representing the spins on the lattice
```

```
    """
    # A hot start means start with +/- 1 with a 50-50 chance on each site
    if is_hot:
        s_local = []
        for i in range(N):
            rdm_num = uniform(0, 1)
            if rdm_num > 0.5:
                s_local.append(1)
            else:
                s_local.append(-1)
    # A cold start means start with 1s on all sites
    else:
        s_local = [1] * N
    return s_local

def get_neighbours_index(N):
    """
    :param N: (int) N = L x L, total number of sites
    :return: (dict) containing as keys the index of the site on the lattice and as values a list containing the indexes
    of its neighbours
    """
    neighbours_dict = {}
    L = int(N**(1/2))
    for i in range(N):
        # store index of neighbours in the values for each node (key, i) in the lattice
        # in the form left, right, top, bottom with periodic boundary conditions
        if i % L == 0:
            left = i + L - 1
        else:
            left = i - 1
        if (i + 1) % L == 0:
            right = i - L + 1
        else:
            right = i + 1
        if i - L < 0:
            top = i - L + N
        else:
            top = i - L
        if i + L >= N:
            bottom = i + L - N
        else:
            bottom = i + L

        neighbours_dict[i] = [left, right, top, bottom]

    return neighbours_dict

def get_energy_difference(index_to_flip, s, neighbours_dictionary):
    """
    :param index_to_flip: (int) the site index to consider flipping its spin
    :param s: (list) spin list of the lattice
    :param neighbours_dictionary: (dict) holds indexes for each site's neighbours
    :return: (float) Total energy change of changing that site's spin
    """
    sum_of_neighbours = 0
    for neighbour_index in neighbours_dictionary[index_to_flip]:
        sum_of_neighbours += s[neighbour_index]
    total_change = 2*s[index_to_flip]*sum_of_neighbours # Works out from the Hamiltonian of the before and after states
    return total_change

def metropolis(dE_4, dE_8, N, s, neighbours_dictionary):
    """
    The metropolis algorithm as a markov chain monte carlo simulation
```

```
    algorithm that modifies the spin state of the
    lattice and gives a new state by choosing N (= L x L) sites at random and checking through the energy if it will
    flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the spin will be flipped and the numbers 4, 8
    represent the corresponding change in energy so that we don't calculate many times an exponential term
    :param dE_4: (float) Probability for energy change of +4
    :param dE_8: (float) Probability for energy change of +8
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (void) does not return anything but changes the state of s
    """
    for i in range(N):
        site_index = randint(0,N-1)
        dE = get_energy_difference(site_index, s, neighbours_dictionary)
        rdm_num = uniform(0, 1)
        if dE <= 0:
            s[site_index] *= -1
        elif dE == 4:
            if dE_4 > rdm_num:
                s[site_index] *= -1
            else:
                continue
        elif dE == 8:
            if dE_8 > rdm_num:
                s[site_index] *= -1
            else:
                continue

def get_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Total magnetisation
    """
    magnetisation_total = 0
    for i in range(N):
        magnetisation_total += s[i]
    return magnetisation_total

def get_energy(N, s, neighbours_dictionary):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :param neighbours_dictionary: (dict) holds indexes for neighbours
    :return: (float) Total energy through the Hamiltonian
    """
    sum1 = 0
    sum2 = 0
    for i in range(N):
        for j in range(number_of_neighbours):
            sum1 += s[i]*s[neighbours_dictionary[i][j]]
        if H != 0:
            sum2 = get_magnetisation(N, s)
    total_energy = (-J*sum1 - mu*H*sum2)/2
    return total_energy

def get_average_magnetisation(N, s):
    """
    :param N: (int) total number of sites
    :param s: (list) spin list
    :return: (float) Magnetisation per site
    """
    return abs(get_magnetisation(N, s))/N

def simulation():
```

```
155
156     with open('deep_ising_data.txt', 'w') as f:
157         # Temperature values in Kelvin
158         T_min = 1.8
159         T_max = 6
160         N = L**2
161         s = get_spin_list(hot_start, N)
162         neighbours_dictionary = get_neighbours_index(N)
163         number_of_runs = 50000
164         configurations = []
165         for i in range(number_of_runs):
166             T = round(random.uniform(T_min, T_max), 2)
167             b = 1 / T  # Constant: 1 / temperature
168             dE_4 = exp(-4 * b)  # Probability for energy change of +4
169             dE_8 = exp(-8 * b)  # Probability for energy change of +8
170             for j in range(nsweeps):
171                 metropolis(dE_4, dE_8, N, s, neighbours_dictionary)
172             configurations.append(s + [T])
173             print(f'Temp: {T} | run: {i} done')
174             str_temp = ','.join(map(str, s+[T]))
175             f.write(f'{str_temp}\n')
176
177 # Uncomment only to generate new data
178 # simulation()
179
180 def get_model():
181     model = Sequential()
182     model.add(Dense(L, activation='sigmoid', input_shape=(L*L,)))
183     model.add(Dense(1, activation='relu'))
184     model.compile(loss='mean_squared_error', optimizer='Adam')
185     return model
186
187 def get_data():
188
189     with open('deep_ising_data.txt', 'r') as f:
190         X = []
191         Y = []
192         line = f.readline()
193         while line:
194             configuration = line.strip().split(',')
195             X.append([int(configuration[:-1][i]) for i in range(len(
    configuration[:-1]))])
196             Y.append(float(configuration[-1]))
197             line = f.readline()
198         X_train = X[:int(len(X)*0.8)]
199         X_test = X[int(len(X)*0.8):]
200         Y_train = Y[:int(len(Y)*0.8)]
201         Y_test = Y[int(len(Y)*0.8):]
202
203     return np.array(X_train), np.array(X_test), np.array(Y_train), np.array(
    Y_test)
204
205 model = load_model('ising_network.h5')
206 # X_train, X_test, Y_train, Y_test = get_data()
207 # model = get_model()
208 # history = model.fit(X_train, Y_train, validation_data=(X_test, Y_test),
    epochs = 20)
209 # plt.plot(history.history['loss'], color = 'k')
210 # plt.plot(history.history['val_loss'], color = 'r')
211 # plt.legend(['MSE', 'Validation MSE'])
212 # plt.title('Neural network prediction performance on temperature')
213 # plt.xlabel('Epoch')
214 # plt.ylabel('Mean squared error')
215 # plt.savefig('deep_ising_mse.pdf', bbox_inches = 'tight')
216 # plt.show()
217 # model.save('ising_network.h5')
218 # model.summary()
219
```

```
def thermalize(T):
    N = L**2
    s = get_spin_list(hot_start, N)
    neighbours_dictionary = get_neighbours_index(N)
    b = 1 / T  # Constant: 1 / temperature
    dE_4 = exp(-4 * b)  # Probability for energy change of +4
    dE_8 = exp(-8 * b)  # Probability for energy change of +8
    for _ in range(nsweeps):
        metropolis(dE_4, dE_8, N, s, neighbours_dictionary)

    return np.array([s])

# predictions = [round(float(model.predict(thermalize(1.9))), 2) for _ in
    range(5000)]
# plt.axvline(x = 1.9, linestyle = '--', color = 'r')
# plt.hist(predictions, color = 'k', bins = 59)
# plt.title('Histogram for DNN prediction on T = 1.9')
# plt.ylabel('Frequency of prediction')
# plt.xlabel('Temperature')
# plt.savefig('deep_histogram2.pdf', bbox_inches = 'tight')
# plt.show()
```