

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]

$$M = \sum_{i=0}^{N-1} s_i \quad (1)$$

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]

$$A(\tau) = \langle (M(t) - \langle M(t) \rangle) \times (M(t + \tau) - \langle M(t + \tau) \rangle) \rangle \quad (2)$$

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

To determine the time lag τ_e over which the autocorrelation 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

$$\tau_e \sim L^z \quad (4)$$

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.

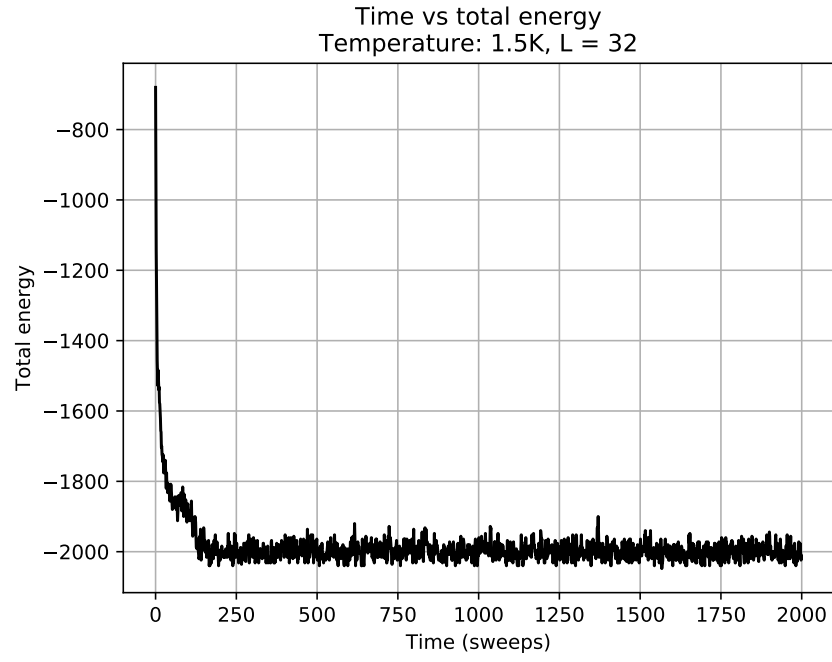


Figure 1: With a hot start the system takes around 250 sweeps at $T = 1.5 J/k_B$ to thermalise.

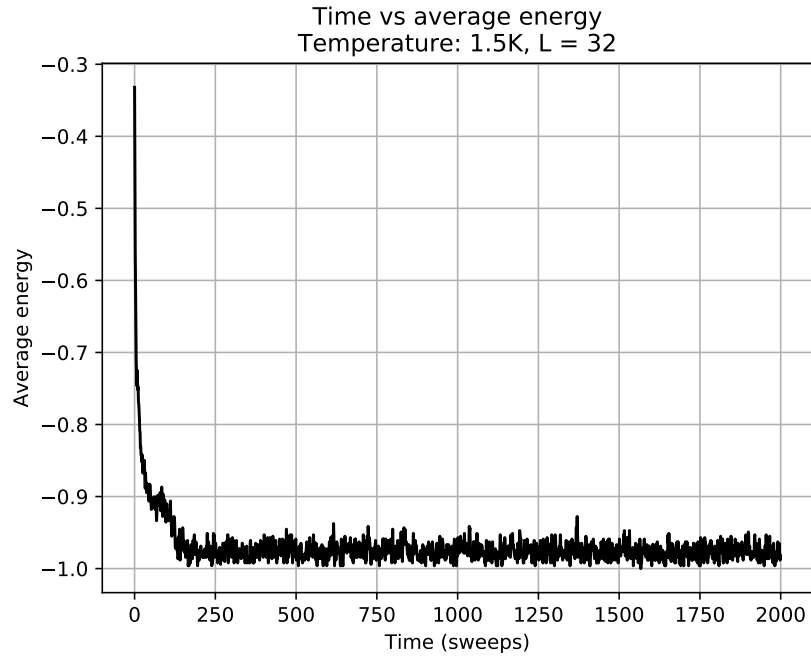


Figure 2: The maximum magnitude of energy per link is 1 and thermalisation time is the same as for the total energy.

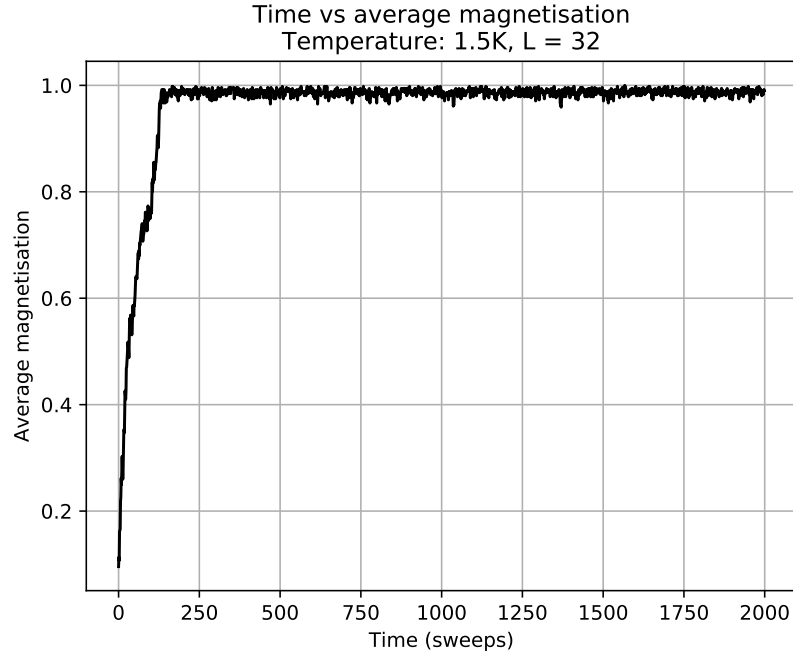


Figure 3: The average magnetisation is the magnetisation per site and its maximum magnitude is 1.

As we approach T_c , the total magnetisation fluctuations σ_M become bigger and for $T \rightarrow 0$ J/k_B , $\sigma_M \rightarrow 0$. 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 .

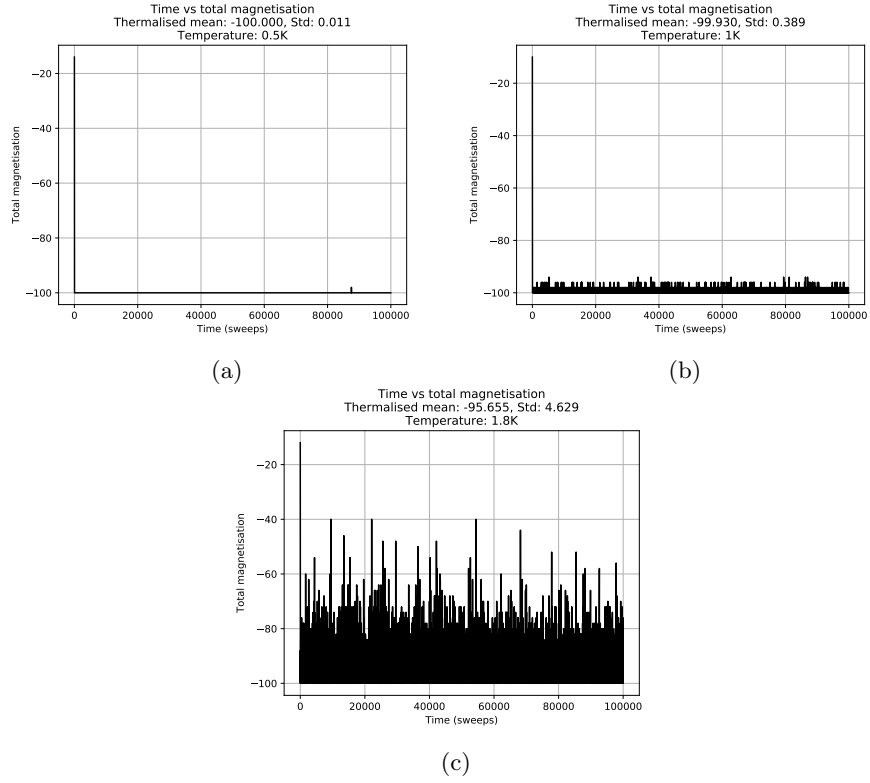


Figure 4: Total magnetisation vs time for $L = 10$ and a hot start. (a): $T = 0.5 \text{ J}/k_B$, (b): $1 \text{ J}/k_B$, (c): $1.8 \text{ J}/k_B$ with mean -100.000, -99.930, -95.655 and standard deviation 0.011, 0.389, 4.629 respectively.

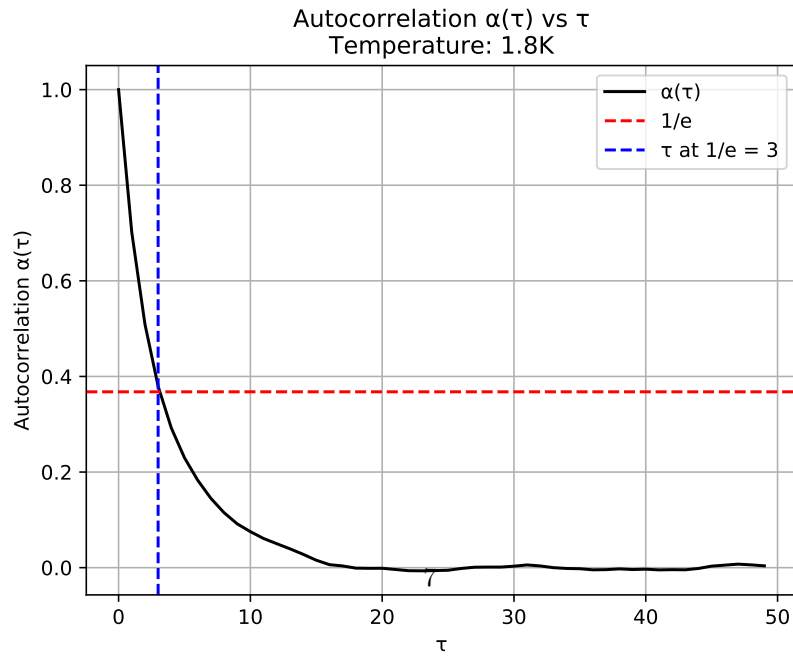


Figure 5: Finding τ_e which characterises the time scale for decorrelation. The plot shows the process for $T = 1.8 \text{ J}/k_B$ with $L = 10$ and a hot start.

L	T (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.

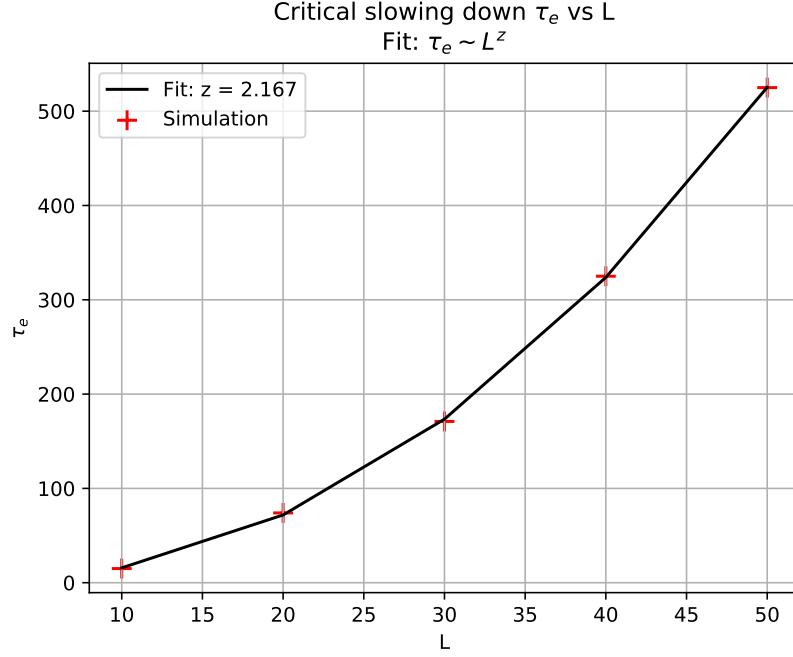


Figure 6: From the fitting of the points shown in this plot, the z exponent in equation (4) was found to be $z = 2.167 \pm 0.037$. The standard deviation is found from the covariance matrix of the fitting algorithm.

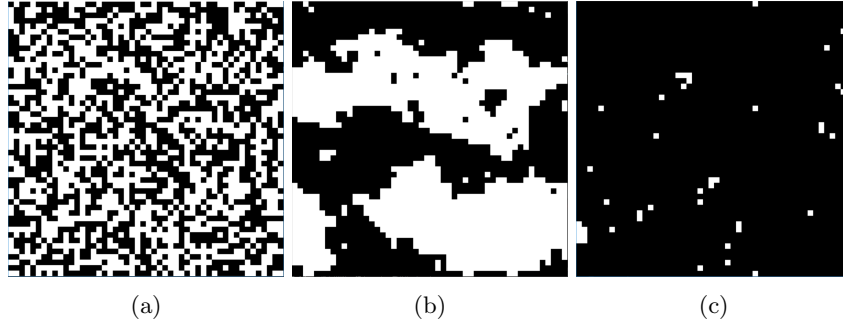


Figure 7: Evolution of cellular automata [5] under the Metropolis algorithm rules. This is the schematic representation of thermalisation at $T = 1.7$ with a hot start. (a) Hot start of randomly oriented spins, (b) After 100 Metropolis sweeps large clusters form, (c) Finally the state has thermalised 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

$$C = \frac{\sigma_E^2}{k_B T^2} \quad (5)$$

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

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

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.

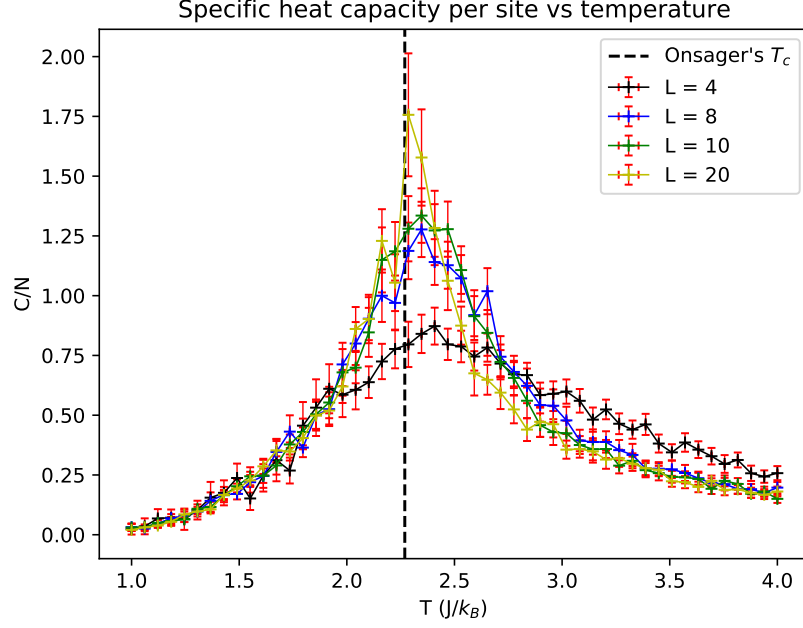


Figure 8: Heat capacity per site vs temperature for different lattice dimensions. As the dimension is increased the sharp change of behaviour near T_c becomes more clear. The error near T_c is always larger and the heat capacity tends to 0 as the temperature tends to 0 and ∞ .

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 $T_c = 2.27J/k_B$ giving a 0.29 percentage difference from Onsager's result.

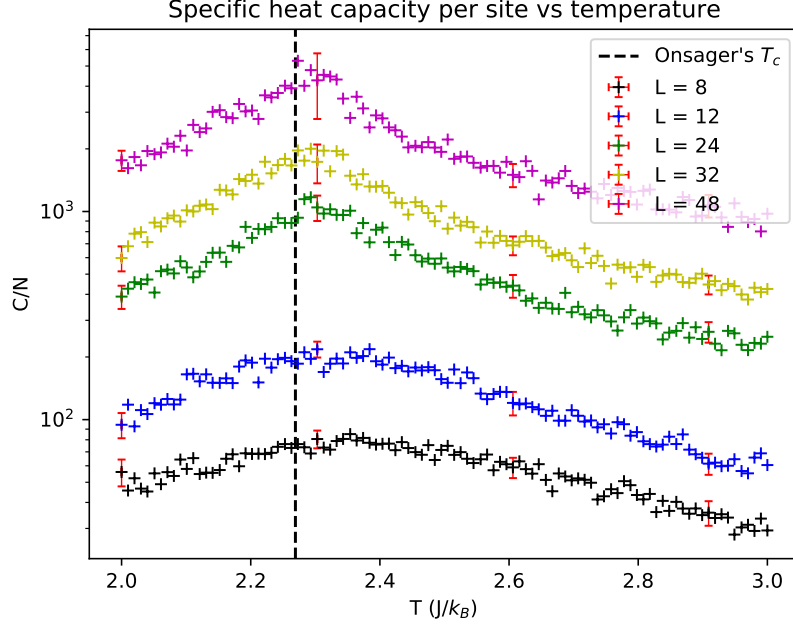


Figure 9: A log scale on the y-axis provides clarity for the behaviour near T_c as the lattice dimension is increased.

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

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

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.

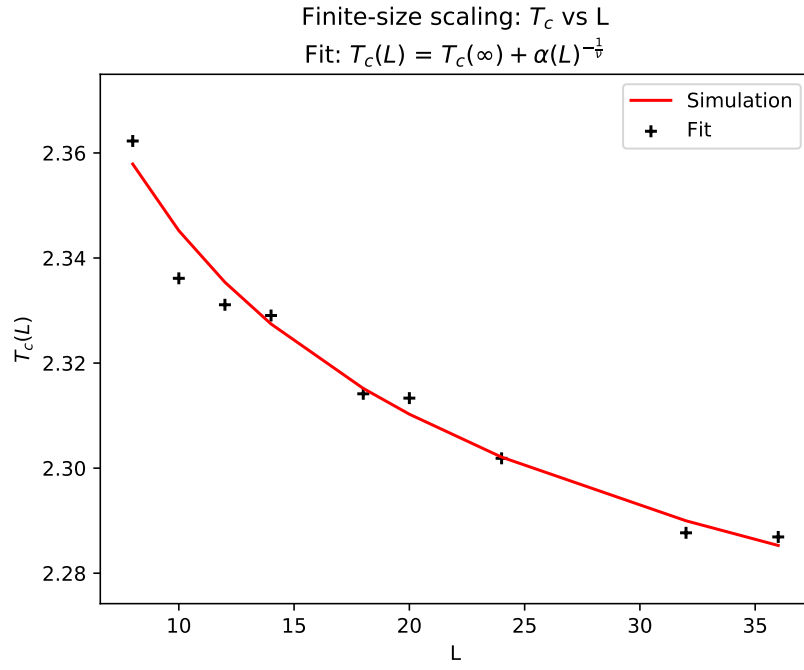


Figure 10: Both the simulation and the fit are seen to converge to Onsager's result at $2.26 J/k_B$.

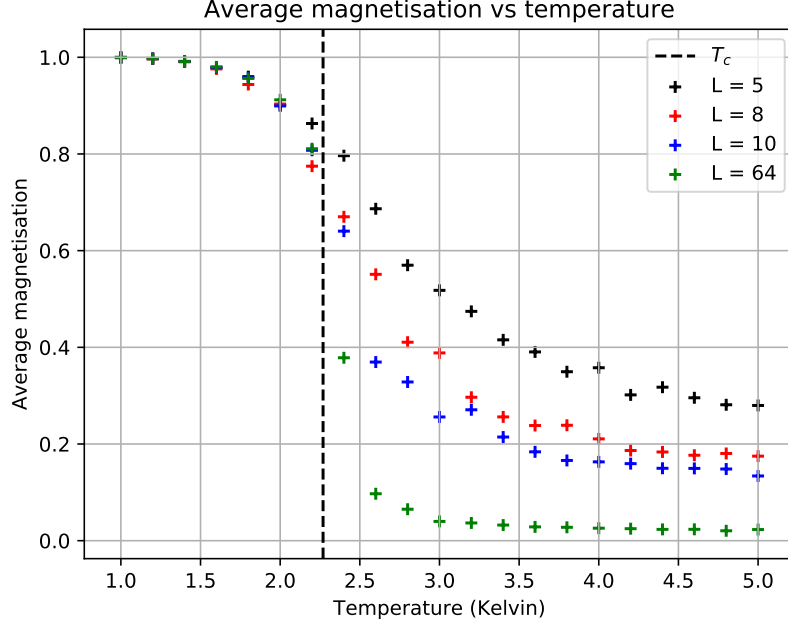


Figure 11: As L is increased, the behaviour approaches the analytical result that predicts the phase transition at $T_c = \frac{2}{\ln(1+\sqrt{2})}$.

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 $\xi \rightarrow \infty$. Above T_c the clusters should dissipate due to the high probability of spin flips and below T_c the typical cluster size $\rightarrow N$, 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 $T \rightarrow \infty$ the cluster size $\rightarrow 1$. The

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

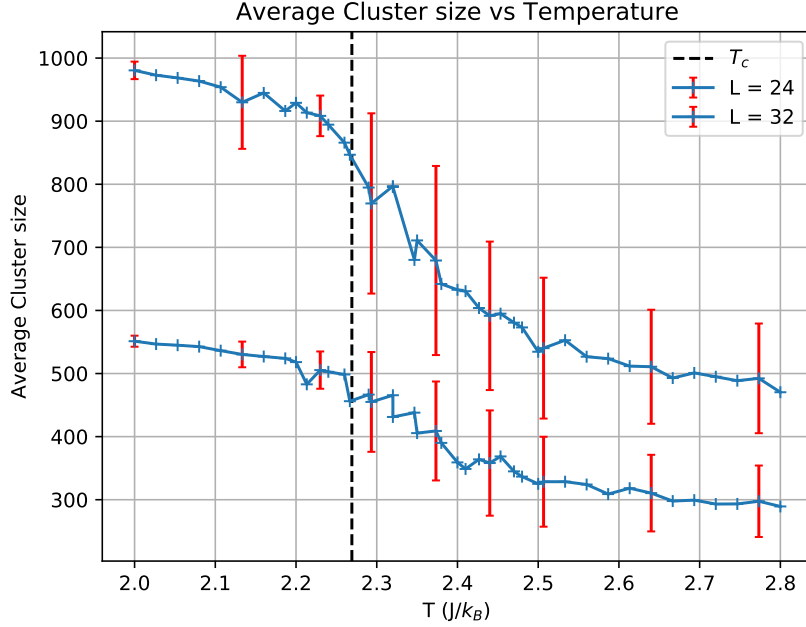


Figure 12: Top line has $L = 32$ and bottom line has $L = 24$. Around T_c , clusters stop dissipating as T increases and the errors are largest due to the large number of possible configurations.

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.

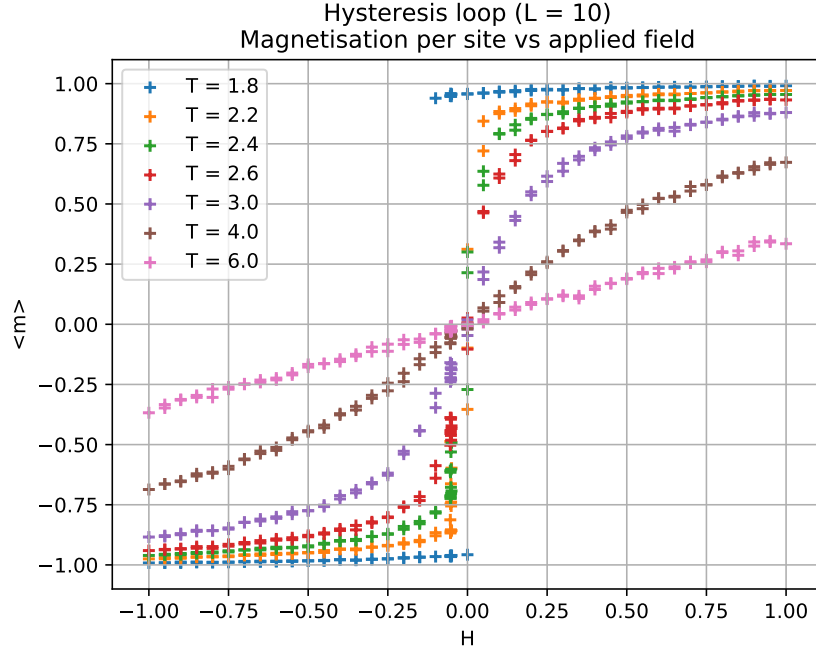


Figure 13: Average magnetisation vs H at different temperatures below and above T_c . Paramagnetism occurs at large $T > T_c$ and hysteresis loops form below T_c .

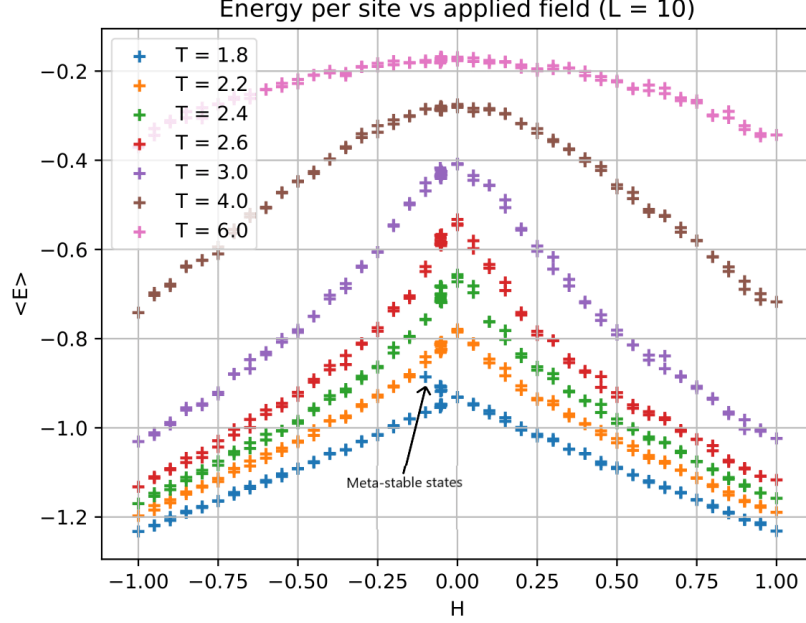


Figure 14: Average energy vs H , from a non-smooth function below T_c with meta-stable states, to a continuous function above T_c with no meta-stable states.

3.6 Magnetic susceptibility using bootstrap

Code used: susceptibility.py

3.6.1 Theory & Methods

The magnetic susceptibility is given by [3]

$$\chi = \frac{\sigma_M^2}{k_B T^2} \quad (8)$$

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 $T \rightarrow 0, \infty$, $\chi \rightarrow 0$. This is because at low T all spins are aligned and the system is ferromagnetic, hence magnetisation is not changing and $\sigma_M \rightarrow 0$. At large T ,

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

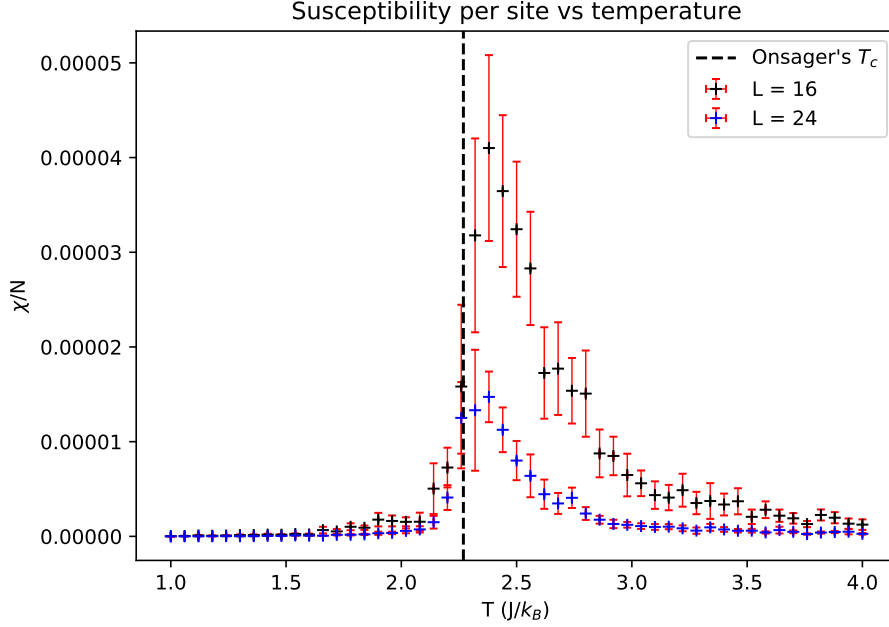


Figure 15: Susceptibility per site vs temperature for different lattice dimensions. As L is increased, susceptibility takes smaller values and the peak approaches Onsager's analytical T_c .

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:

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

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 $J \rightarrow 0$ and as L is increased the transition becomes sharper and well-defined.

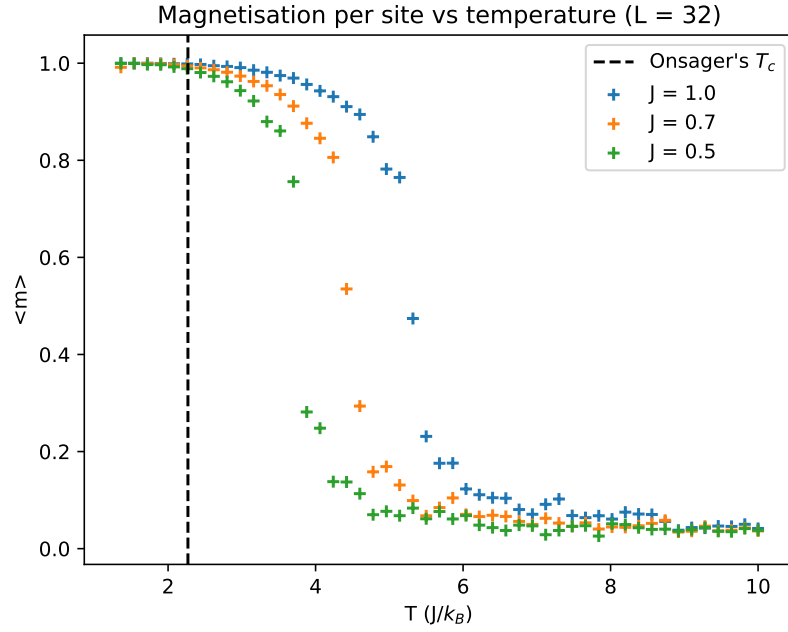


Figure 16: Next-nearest neighbour interactions cause T_c to increase and varies with the degree of the exchange energy magnitude. The dashed line is the critical temperature when next-nearest neighbour interactions are turned off.

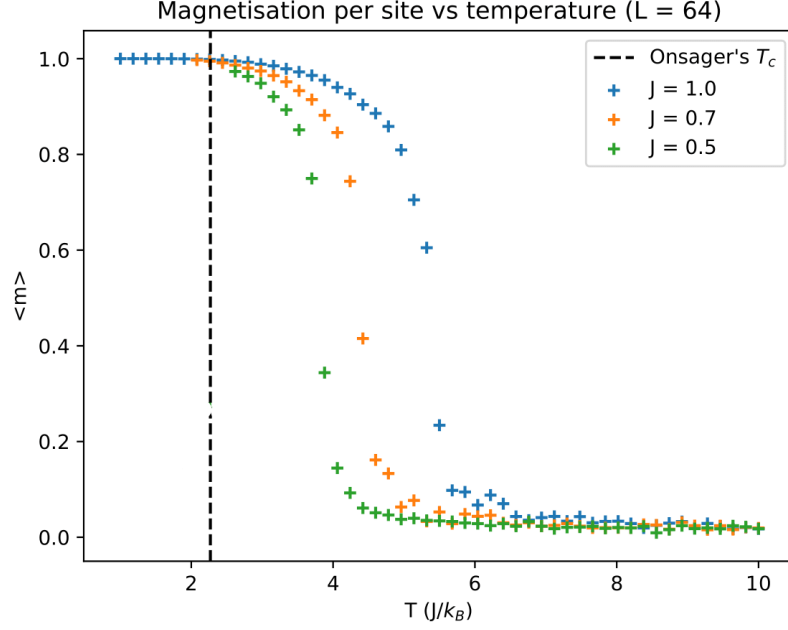


Figure 17: The next-nearest neighbour phase transition becomes sharper as $L \rightarrow \infty$. For $J = 1$, T_c is around $5.5 J/k_B$, which is more than double from the model investigated in the previous sections, shown here with the dashed line.

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.

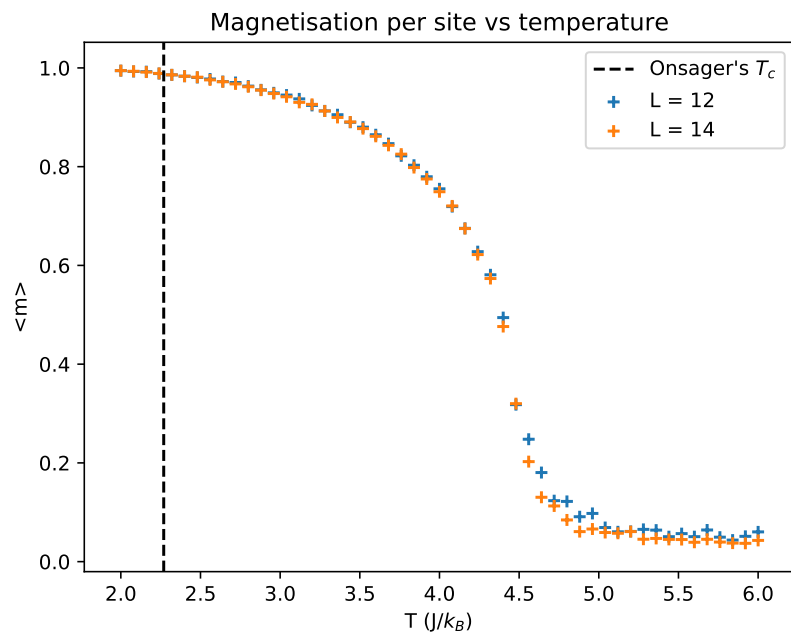


Figure 18: Average magnetisation vs temperature for the 3D Ising model, showing a T_c around $4.5 J/k_B$. The dashed line is the critical temperature in 2D.

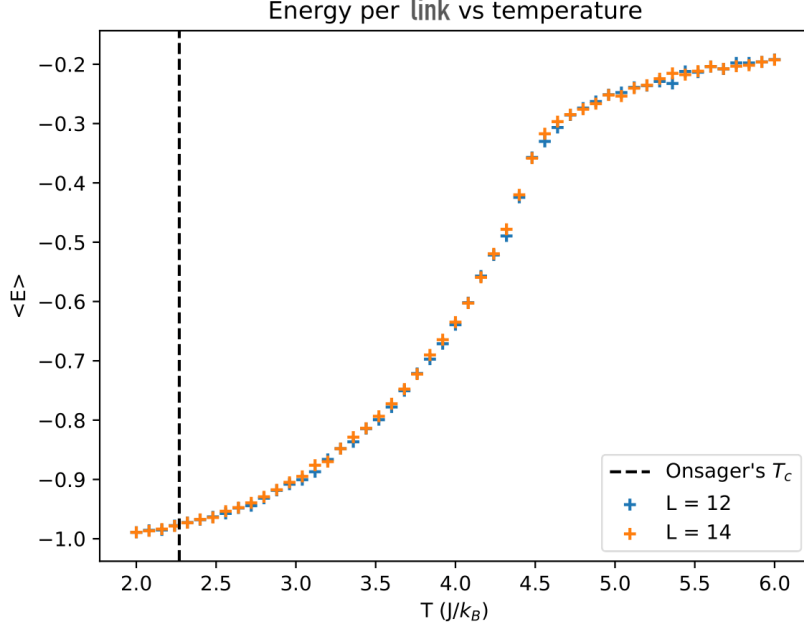


Figure 19: The energy vs temperature for the 3D case again indicates a higher T_c and as $T \rightarrow \infty$, energy $\rightarrow 0$, since at high T it is more likely that half of the neighbours are aligned and the other half are anti-aligned.

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 $L \times L$ 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.

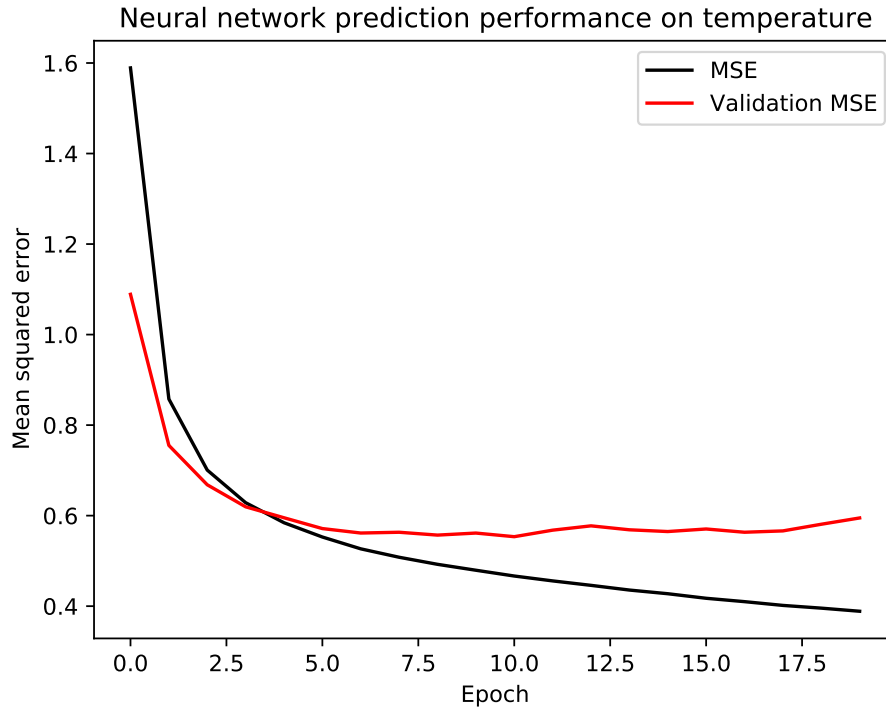


Figure 20: After 5 epochs, the validation starts diverging which is an indication of over-fitting.

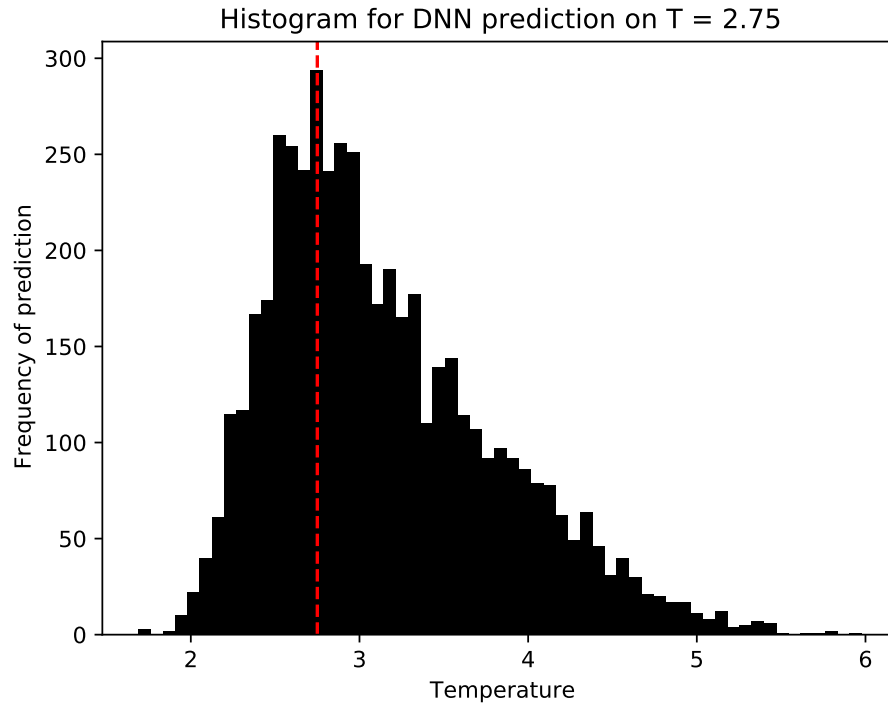


Figure 21: The prediction on 5000 configurations all with $T = 2.75$ shows gaussian behaviour as suggested by the central limit theorem with a mean at the correct temperature. The red dashed line is at $T = 2.75$.

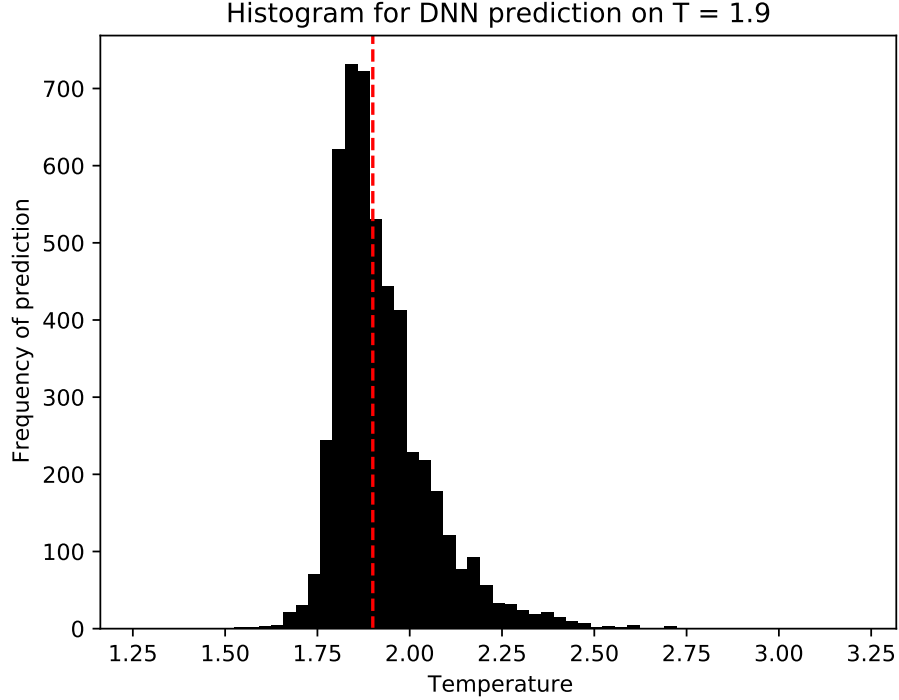


Figure 22: The prediction on 5000 configurations all with $T = 1.9$ shows gaussian behaviour as suggested by the central limit theorem with a mean at the correct temperature. Below T_c , all configurations are almost identical - hence the spread is less than in Figure 21 - and the mean is again centered on the correct temperature. It is possible that the model has picked up and learned on small fluctuations occurring 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 $T_c = 2.2758 \pm 0.0096 \text{ J}/k_B$. Finite size scaling gave $T_c(\infty) = 2.264 \pm 0.152 \text{ J}/k_B$.
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 $T \rightarrow \infty$ the average cluster size decayed to one. Above T_c there is no hysteresis and no metastable states and vice versa. As $T \rightarrow \infty$, 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

avg_mag.py	29
ising_ca_reader.py	32
avg_mag_table.py	33
bootstrap.py	36
bootstrap_susceptibility.py	37
correlation.py	38
correlation_table.py	41
domains.py	46
heat_capacity.py	51
heat_capacity_plotter.py	56
hysteresis.py	58
nearest_neighbours.py	63
susceptibility.py	67
3d_lattice.py	71
deep_ising_temp.py	74

Code file: avg_mag.py

```
1 from random import uniform, randint
2 import random
3 from math import exp
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import pandas as pd
7
8 # Global variables
9 L = 10 # Lattice dimension
10 N = L**2 # Total number of nodes in lattice
11 J = 1 # Exchange energy
12 H = -2.5 # Applied magnetic field strength
13 mu = 1 # Magnetic moment
14 M = 0 # Total magnetisation
15 T = 0.9 # Temperature in Kelvin
16 b = 1/T # Constant: 1 / temperature
17 dE_4 = exp(-4*b) # Probability for energy change of +4
18 dE_8 = exp(-8*b) # Probability for energy change of +8
19 number_of_neighbours = 4
20 random_seed = 31 # Fix random seed in spin initialization for reproducibility
21 hot_start = True # Initialize with hot start
22 nsweeps = 2000 # Number of sweeps
23
24 def get_spin_list(is_hot):
25     """
26     :param is_hot: (bool) If is_hot is true make hot start otherwise make
27     cold start
28     :return: (list) containing +/- 1 representing the spins on the lattice
29     """
30     random.seed(random_seed)
31     # A hot start means start with +/- 1 with a 50-50 chance on each site
32     if is_hot:
33         s_local = []
34         for i in range(N):
```

```

34         rdm_num = uniform(0, 1) # Get a float between 0 and 1 from a
        uniform distribution
35         if rdm_num > 0.5:
36             s_local.append(1)
37         else:
38             s_local.append(-1)
39         # A cold start means start with 1s on all sites
40     else:
41         s_local = [1] * N # Creates a list of N (+1)'s
42     return s_local
43
44 def get_neighbours_index():
45     """
46     :return: (dict) containing as keys the index of the site on the lattice
47             and as values a list containing the indexes
48             of its neighbours
49     """
50     neighbours_dict = {} # Index of site -> indexes of neighbours of that
51                          # site (key -> value)
52     for i in range(N):
53         # store index of neighbours in the values for each node (key, i) in
54         # the lattice
55         # in the form left, right, top, bottom with periodic boundary
56         # conditions
57         if i % L == 0:
58             left = i + L - 1
59         else:
60             left = i - 1
61         if (i + 1) % L == 0:
62             right = i - L + 1
63         else:
64             right = i + 1
65         if i - L < 0:
66             top = i - L + N
67         else:
68             top = i - L
69         if i + L >= N:
70             bottom = i + L - N
71         else:
72             bottom = i + L
73
74         neighbours_dict[i] = [left, right, top, bottom]
75
76     return neighbours_dict
77
78 s = get_spin_list(hot_start) # Initialise a lattice
79 neighbours_dictionary = get_neighbours_index() # Initialise the dictionary
80 holding the neighbours
81
82 def get_energy_difference(index_to_flip):
83     """
84     :param index_to_flip: (int) the site index to consider flipping its spin
85     :return: (float) Total energy change of changing that site's spin
86     """
87     sum_of_neighbours = 0
88     for neighbour_index in neighbours_dictionary[index_to_flip]:
89         sum_of_neighbours += s[neighbour_index]
90     total_change = 2*s[index_to_flip]*sum_of_neighbours + 2*s[index_to_flip]*
91     mu*H
92     return total_change
93
94 def metropolis():
95     """
96     The metropolis algorithm as a markov chain monte carlo simulation
97     algorithm that modifies the spin state of the
98     lattice and gives a new state by choosing N (= L x L) sites at random and
99     checking through the energy if it will
100     flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the

```

```

93     spin will be flipped and the numbers 4, 8
94     represent the corresponding change in energy so that we don't calculate
95     many times an exponential term
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)
101         rdm_num = uniform(0, 1)
102         if dE <= 0:
103             s[site_index] *= -1
104         elif dE == 4:
105             if dE_4 > rdm_num:
106                 s[site_index] *= -1
107             else:
108                 continue
109         elif dE == 8:
110             if dE_8 > rdm_num:
111                 s[site_index] *= -1
112             else:
113                 continue
114
115 def get_magnetisation():
116     """
117     :return: (float) Total magnetisation
118     """
119     magnetisation_total = 0
120     for i in range(N):
121         magnetisation_total += s[i]
122     return magnetisation_total
123
124 def get_energy():
125     """
126     :return: (float) Total energy through the Hamiltonian
127     """
128     sum1 = 0
129     sum2 = 0
130     for i in range(N):
131         for j in range(number_of_neighbours):
132             sum1 += s[i]*s[neighbours_dictionary[i][j]]
133             if H != 0:
134                 sum2 = get_magnetisation()
135     total_energy = (-J*sum1 - mu*H*sum2)/2
136     return total_energy
137
138 def get_average_energy():
139     """
140     :return: (float) Average energy through the Hamiltonian
141     """
142     sum1 = 0
143     sum2 = 0
144     for i in range(N):
145         for j in range(number_of_neighbours):
146             sum1 += s[i]*s[neighbours_dictionary[i][j]]
147             if H != 0:
148                 sum2 = get_magnetisation()
149     total_energy = (-J*sum1 - mu*H*sum2)/2
150     return total_energy/(2*N)
151
152 def get_average_magnetisation():
153     """
154     :return: (float) Magnetisation per site
155     """
156     return get_magnetisation()/N
157
158 def simulation():
159     """
160     Goes through all sweeps modifying the state with the metropolis function

```

```

159     and logging total
magnetisation and energy. Then calculates the mean and standard deviation
    of the total magnetisation after
160 thermalisation to quantify its fluctuations. The it plots time vs total
and average energy and time vs magnetisation
: return: (void)
161 """
162 total_magnetisation = []
163 total_energy = []
164 nsweeps_values = np.arange(0, nsweeps)
165
166 # Evolve the system and save magnetisation and energy for each new state
167 for i in range(nsweeps):
168     metropolis()
169     total_magnetisation.append(get_magnetisation())
170     total_energy.append(get_energy())
171
172 # Quantify fluctuations after thermalisation of 1000 sweeps
173 mean = np.average(total_magnetisation[1000:])
174 standard_deviation = np.std(total_magnetisation[1000:])
175
176 plt.plot(nsweeps_values, total_magnetisation, color = 'k')
177 plt.xlabel('Time (sweeps)')
178 plt.ylabel('Total magnetisation')
179 plt.title(f'Time vs total magnetisation\nThermalised mean: {mean:.3f},
180 Std: {standard_deviation:.3f}\nTemperature: {T}K, L = {L}')
181 plt.grid(True)
182 # plt.savefig(f'time_vs_total_mag_{T}.pdf', bbox = 'tight')
183 plt.show()
184
185 plt.plot(nsweeps_values, np.array(total_magnetisation)/N, color='k')
186 plt.xlabel('Time (sweeps)')
187 plt.ylabel('Average magnetisation')
188 plt.title(
189     f'Time vs average magnetisation\nTemperature: {T}K, L = {L}')
190 plt.grid(True)
191 # plt.savefig(f't_vs_avg_mag_{T}.pdf', bbox = 'tight')
192 plt.show()
193
194 plt.plot(nsweeps_values, total_energy, color = 'k')
195 plt.xlabel('Time (sweeps)')
196 plt.ylabel('Total energy')
197 plt.title(f'Time vs total energy\nTemperature: {T}K, L = {L}')
198 plt.grid(True)
199 # plt.savefig(f'time_vs_total_energy_{T}.pdf', bbox = 'tight')
200 plt.show()
201
202 average_energy = np.array(total_energy)/(2*N)
203 plt.plot(nsweeps_values, average_energy, color='k')
204 plt.xlabel('Time (sweeps)')
205 plt.ylabel('Average energy')
206 plt.title(f'Time vs average energy\nTemperature: {T}K, L = {L}')
207 plt.grid(True)
208 # plt.savefig(f'time_vs_avg_energy_{T}.pdf', bbox = 'tight')
209 plt.show()
210
211 simulation()

```

Code file: ising_ca_reader.py

```

1 import pyglet
2 import numpy as np
3
4 class Lattice:
5
6     def __init__(self, window_width, window_height, cell_size):
7         self.grid_width = int(window_width / cell_size)
8         self.grid_height = int(window_height / cell_size)

```



```

9         self.cell_size = cell_size
10        self.configurations = self.get_sites()
11        self.config_num = 0
12
13    def get_sites(self):
14        with open('ising-ca.txt', 'r') as file:
15            configurations = []
16            line = file.readline()
17            while line:
18                configuration = line.strip().split(' ')
19                configurations.append(configuration)
20                line = file.readline()
21            return configurations
22
23    def draw(self):
24        try:
25            configuration = self.configurations[self.config_num]
26        except IndexError:
27            print('The simulation is over!')
28            quit()
29        L = int(np.sqrt(len(configuration)))
30        for i in range(len(configuration)):
31            row = int(i/L)
32            col = i % L
33            if int(configuration[i]) == 1:
34                square_coords = (row*self.cell_size, col*self.cell_size,
35                                row*self.cell_size +
36                                self.cell_size,
37                                row*self.cell_size + self.cell_size, col*
38                                self.cell_size,
39                                row*self.cell_size+self.cell_size, col*self.
40                                cell_size+self.cell_size)
41                pyglet.graphics.draw_indexed(4, pyglet.gl.GL_TRIANGLES, [0,
42                                1, 2, 1, 2, 3], ('v2i', square_coords))
43
44    class Window(pyglet.window.Window):
45
46    def __init__(self):
47        super().__init__(600, 600)
48        self.lattice = Lattice(self.get_size()[0], self.get_size()[1], int(
49        self.get_size()[0]/50))
50        pyglet.clock.schedule_interval(self.update, 1.0/25.0)
51
52    def on_draw(self):
53        self.clear()
54        self.lattice.draw()
55
56    def update(self, dt):
57        self.lattice.config_num += 1
58
59    if __name__ == '__main__':
60        window = Window()
61        pyglet.app.run()

```

Code file: avg_mag_table.py

```

1 from random import uniform, randint
2 import random
3 from math import exp
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import pandas as pd
7
8 # Global variables
9 J = 1 # Exchange energy
10 H = 0 # Applied magnetic field strength
11 mu = 1 # Magnetic moment
12 M = 0 # Total magnetisation

```

```

13 number_of_neighbours = 4
14 random_seed = 10 # Fix random seed in spin initialization for reproducibility
15 hot_start = False # Initialize with hot start or not
16 nsweeps = 1000 # Number of sweeps
17
18 def get_spin_list(is_hot, N):
19     """
20     :param is_hot: (bool) If is_hot is true make hot start otherwise make
21     cold start
22     :param N: (int) N = L x L, total number of sites
23     :return: (list) containing +/- 1 representing the spins on the lattice
24     """
25     random.seed(random_seed)
26     # A hot start means start with +/- 1 with a 50-50 chance on each site
27     if is_hot:
28         s_local = []
29         for i in range(N):
30             rdm_num = uniform(0, 1)
31             if rdm_num > 0.5:
32                 s_local.append(1)
33             else:
34                 s_local.append(-1)
35     # A cold start means start with 1s on all sites
36     else:
37         s_local = [1] * N
38     return s_local
39
40 def get_neighbours_index(N):
41     """
42     :param N: (int) N = L x L, total number of sites
43     :return: (dict) containing as keys the index of the site on the lattice
44     and as values a list containing the indexes
45     of its neighbours
46     """
47     neighbours_dict = {}
48     L = int(N**(1/2))
49     for i in range(N):
50         # store index of neighbours in the values for each node (key, i) in
51         # the lattice
52         # in the form left, right, top, bottom with periodic boundary
53         conditions
54         if i % L == 0:
55             left = i + L - 1
56         else:
57             left = i - 1
58         if (i + 1) % L == 0:
59             right = i - L + 1
60         else:
61             right = i + 1
62         if i - L < 0:
63             top = i - L + N
64         else:
65             top = i - L
66         if i + L >= N:
67             bottom = i + L - N
68         else:
69             bottom = i + L
70
71         neighbours_dict[i] = [left, right, top, bottom]
72
73     return neighbours_dict
74
75 def get_energy_difference(index_to_flip, s, neighbours_dictionary):
76     """
77     :param index_to_flip: (int) the site index to consider flipping its spin
78     :param s: (list) spin list of the lattice
79     :param neighbours_dictionary: (dict) holds indexes for each site's
80     neighbours

```

```

76 :return: (float) Total energy change of changing that site's spin
77 """
78 sum_of_neighbours = 0
79 for neighbour_index in neighbours_dictionary[index_to_flip]:
80     sum_of_neighbours += s[neighbour_index]
81 total_change = 2*s[index_to_flip]*sum_of_neighbours # Works out from the
82 Hamiltonian of the before and after states
83 return total_change
84
85 def metropolis(dE_4, dE_8, N, s, neighbours_dictionary):
86     """
87     The metropolis algorithm as a markov chain monte carlo simulation
88     algorithm that modifies the spin state of the
89     lattice and gives a new state by choosing N (= L x L) sites at random and
90     checking through the energy if it will
91     flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
92     spin will be flipped and the numbers 4, 8
93     represent the corresponding change in energy so that we don't calculate
94     many times an exponential term
95     :param dE_4: (float) Probability for energy change of +4
96     :param dE_8: (float) Probability for energy change of +8
97     :param N: (int) total number of sites
98     :param s: (list) spin list
99     :param neighbours_dictionary: (dict) holds indexes for neighbours
100     :return: (void) does not return anything but changes the state of s
101     """
102     for i in range(N):
103         site_index = randint(0,N-1)
104         dE = get_energy_difference(site_index, s, neighbours_dictionary)
105         rdm_num = uniform(0, 1)
106         if dE <= 0:
107             s[site_index] *= -1
108         elif dE == 4:
109             if dE_4 > rdm_num:
110                 s[site_index] *= -1
111             else:
112                 continue
113         elif dE == 8:
114             if dE_8 > rdm_num:
115                 s[site_index] *= -1
116             else:
117                 continue
118
119 def get_magnetisation(N, s):
120     """
121     :param N: (int) total number of sites
122     :param s: (list) spin list
123     :return: (float) Total magnetisation
124     """
125     magnetisation_total = 0
126     for i in range(N):
127         magnetisation_total += s[i]
128     return magnetisation_total
129
130 def get_energy(N, s, neighbours_dictionary):
131     """
132     :param N: (int) total number of sites
133     :param s: (list) spin list
134     :param neighbours_dictionary: (dict) holds indexes for neighbours
135     :return: (float) Total energy through the Hamiltonian
136     """
137     sum1 = 0
138     sum2 = 0
139     for i in range(N):
140         for j in range(number_of_neighbours):
141             sum1 += s[i]*s[neighbours_dictionary[i][j]]
142         if H != 0:
143             sum2 = get_magnetisation(N, s)

```

```

139 total_energy = (-J*sum1 - mu*H*sum2)/2
140 return total_energy
141
142 def get_average_magnetisation(N, s):
143     """
144     :param N: (int) total number of sites
145     :param s: (list) spin list
146     :return: (float) Magnetisation per site
147     """
148     return abs(get_magnetisation(N, s))/N
149
150 def simulation():
151     """
152     The simulation function gathers data for average magnetisation vs
153     temperature for different L dimensions of the
154     lattice. It stores the data in a pandas dataframe and makes a plot of all
155     L in the same average magnetisation vs
156     temperature graph
157     :return: (void)
158     """
159     # Temperature values in Kelvin
160     T_values = np.linspace(1, 5, num = 21)
161     # Dimension values
162     L_values = [5, 8, 10, 64]
163     # Dataframe to store magnetisation values vs temperature for different L
164     dataframe = pd.DataFrame() # T_values, columns = ['Temperature']
165     for L in L_values:
166         N = L**2
167         s = get_spin_list(hot_start, N)
168         neighbours_dictionary = get_neighbours_index(N)
169         average_magnetisation = []
170         for T in T_values:
171             b = 1 / T # Constant: 1 / temperature
172             dE_4 = exp(-4 * b) # Probability for energy change of +4
173             dE_8 = exp(-8 * b) # Probability for energy change of +8
174             tmp_magnetisation = []
175             for i in range(nsweeps):
176                 metropolis(dE_4, dE_8, N, s, neighbours_dictionary)
177                 tmp_magnetisation.append(get_average_magnetisation(N, s))
178             average_magnetisation.append(np.average(tmp_magnetisation[100:]))
179             print(T)
180             dataframe[f'L{L}'] = average_magnetisation
181             print(L, '-----')
182
183     plt.scatter(T_values, dataframe['L5'], color = 'k', marker = '+')
184     plt.scatter(T_values, dataframe['L8'], color = 'r', marker = '+')
185     plt.scatter(T_values, dataframe['L10'], color = 'b', marker = '+')
186     plt.scatter(T_values, dataframe['L64'], color = 'g', marker = '+')
187     plt.axvline(x = 2/np.log(1+np.sqrt(2)), color = 'k', linestyle = '--')
188     plt.xlabel('T (J/$k_B$)')
189     plt.ylabel('Average magnetisation')
190     plt.title('Average magnetisation vs temperature')
191     plt.grid(True)
192     plt.legend(['$T_c$, 'L = 5', 'L = 8', 'L = 10', 'L = 64'])
193     # plt.savefig('avg_mag_vs_temp.pdf', bbox = 'tight')
194     plt.show()
195
196 simulation()

```

Code file: bootstrap.py

```

1 import random
2 import numpy as np
3
4 def get_sigma_energy(pseudo_bin_list):
5     """
6     :param pseudo_bin_list: (list) contains the pseudo data of one bin
7     :return: (float) the standard deviation of the input list

```

```

8     """
9     return np.std(pseudo_bin_list)
10
11 def get_heat_capacity(sigma_energy, T):
12     """
13     :param sigma_energy: (float) the standard deviation of the energy from a
14     bin list
15     :param T: (float) Temperature
16     :return: (float) the specific heat capacity
17     """
18     return (sigma_energy**2) / (T**2)
19
20 def bootstrap(energy_list, n_bins, T, target_tau):
21     """
22     :param target_tau: (int) decorrelation constant
23     :param energy_list: (list) energy for each sweep
24     :param n_bins: (int) number of bins to include in the bootstrap method
25     :param T: (float) Temperature
26     :return: (float, float) 2-tuple of heat capacity and its standard
27     deviation
28     """
29     length = len(energy_list)
30     # Initialise the bin dictionary holding values of lists. These lists will
31     # contain pseudo data of energy
32     bins_dict = {idx: [] for idx in range(n_bins)}
33     # This for loop traverses the bin keys in the dictionary bins_dict
34     for bin_index in range(n_bins):
35         # This for loop runs for the length of the energy list
36         for i in range(length):
37             random_index = random.randint(0, length-1)
38             bins_dict[bin_index].append(energy_list[random_index]) # Append
39             pseudo data to bin list
40
41     sigma_energy_list = [get_sigma_energy(bins_dict[key_bin]) for key_bin in
42     bins_dict]
43     C_list = [get_heat_capacity(sigma_energy_list[i], T) for i in range(len(
44     sigma_energy_list))]
45
46     C_mean = np.average(C_list)
47     sigma_C = np.std(C_list)*np.sqrt(1 + 2*target_tau)
48
49     return C_mean, sigma_C

```

Code file: bootstrap_susceptibility.py

```

1 import random
2 import numpy as np
3
4 def get_sigma_magnetisation(pseudo_bin_list):
5     """
6     :param pseudo_bin_list: (list) contains the pseudo data of one bin
7     :return: (float) the standard deviation of the input list
8     """
9     return np.std(pseudo_bin_list)
10
11 def get_susceptibility(sigma_magnetisation, T):
12     """
13     :param sigma_magnetisation: (float) the standard deviation of the
14     magnetisation from a bin list
15     :param T: (float) Temperature
16     :return: (float) the susceptibility
17     """
18     return (sigma_magnetisation**2) / (T**2)
19
20 def bootstrap(magnetisation_list, n_bins, T, target_tau):
21     """
22     :param target_tau: (int) decorrelation constant
23     :param magnetisation_list: (list) magnetisation for each sweep

```

```

23 :param n_bins: (int) number of bins to include in the bootstrap method
24 :param T: (float) Temperature
25 :return: (float, float) 2-tuple of susceptibility and its standard
        deviation
26 """
27 length = len(magnetisation_list)
28 # Initialise the bin dictionary holding values of lists. These lists will
        contain pseudo data of magnetisation
29 bins_dict = {idx: [] for idx in range(n_bins)}
30 # This for loop traverses the bin keys in the dictionary bins_dict
31 for bin_index in range(n_bins):
32     # This for loop runs for the length of the magnetisation list
33     for i in range(length):
34         random_index = random.randint(0, length-1)
35         bins_dict[bin_index].append(magnetisation_list[random_index]) #
        Append pseudo data to bin list
36
37 sigma_magnetisation_list = [get_sigma_magnetisation(bins_dict[key_bin])
        for key_bin in bins_dict]
38 chi_list = [get_susceptibility(sigma_magnetisation_list[i], T) for i in
        range(len(sigma_magnetisation_list))]
39
40 chi_mean = np.average(chi_list)
41 sigma_chi = np.std(chi_list)*np.sqrt(1 + 2*target_tau)
42
43 return chi_mean, sigma_chi

```

Code file: correlation.py

```

1 from random import uniform, randint
2 import random
3 from math import exp
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import pandas as pd
7
8 # Global variables
9 L = 30 # Lattice dimension
10 N = L**2 # Total number of nodes in lattice
11 J = 1 # Exchange energy
12 H = 0 # Applied magnetic field strength
13 mu = 1 # Magnetic moment
14 M = 0 # Total magnetisation
15 T = 1.0 # Temperature in Kelvin
16 b = 1/T # Constant: 1 / temperature
17 dE_4 = exp(-4*b) # Probability for energy change of +4
18 dE_8 = exp(-8*b) # Probability for energy change of +8
19 number_of_neighbours = 4
20 random_seed = 13 # Fix random seed in spin initialization for reproducability
21 hot_start = True # Initialize with hot start
22 nsweeps = 30000 # Number of sweeps
23
24 def get_spin_list(is_hot):
25     """
26     :param is_hot: (bool) If is_hot is true make hot start otherwise make
        cold start
27     :return: (list) containing +/- 1 representing the spins on the lattice
28     """
29     random.seed(random_seed)
30     # A hot start means start with +/- 1 with a 50-50 chance on each site
31     if is_hot:
32         s_local = []
33         for i in range(N):
34             rdm_num = uniform(0, 1)
35             if rdm_num > 0.5:
36                 s_local.append(1)
37             else:
38                 s_local.append(-1)

```

```

39 # A cold start means start with 1s on all sites
40 else:
41     s_local = [1] * N
42     return s_local
43
44 def get_neighbours_index():
45     """
46     :return: (dict) containing as keys the index of the site on the lattice
47             and as values a list containing the indexes
48             of its neighbours
49     """
50     neighbours_dict = {}
51     for i in range(N):
52         # store index of neighbours in the values for each node (key, i) in
53         # the lattice
54         # in the form left, right, top, bottom with periodic boundary
55         # conditions
56         if i % L == 0:
57             left = i + L - 1
58         else:
59             left = i - 1
60         if (i + 1) % L == 0:
61             right = i - L + 1
62         else:
63             right = i + 1
64         if i - L < 0:
65             top = i - L + N
66         else:
67             top = i - L
68         if i + L >= N:
69             bottom = i + L - N
70         else:
71             bottom = i + L
72
73         neighbours_dict[i] = [left, right, top, bottom]
74
75     return neighbours_dict
76
77 s = get_spin_list(hot_start)
78 neighbours_dictionary = get_neighbours_index()
79
80 def get_energy_difference(index_to_flip):
81     """
82     :param index_to_flip: (int) the site index to consider flipping its spin
83     :return: (float) Total energy change of changing that site's spin
84     """
85     sum_of_neighbours = 0
86     for neighbour_index in neighbours_dictionary[index_to_flip]:
87         sum_of_neighbours += s[neighbour_index]
88     total_change = 2*s[index_to_flip]*sum_of_neighbours # Works out from the
89                                                         # Hamiltonian of the before and after states
90     return total_change
91
92 def metropolis():
93     """
94     The metropolis algorithm as a markov chain monte carlo simulation
95     algorithm that modifies the spin state of the
96     lattice and gives a new state by choosing N (= L x L) sites at random and
97     checking through the energy if it will
98     flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
99     spin will be flipped and the numbers 4, 8
100     represent the corresponding change in energy so that we don't calculate
101     many times an exponential term
102     :return: (void) does not return anything but changes the state of s
103     """
104     for i in range(N):
105         site_index = randint(0,N-1)
106         dE = get_energy_difference(site_index)

```

```

99         rdm_num = uniform(0, 1)
100         if dE <= 0:
101             s[site_index] *= -1
102         elif dE == 4:
103             if dE_4 > rdm_num:
104                 s[site_index] *= -1
105             else:
106                 continue
107         elif dE == 8:
108             if dE_8 > rdm_num:
109                 s[site_index] *= -1
110             else:
111                 continue
112
113 def get_magnetisation():
114     """
115     :return: (float) Total magnetisation
116     """
117     magnetisation_total = 0
118     for i in range(N):
119         magnetisation_total += s[i]
120     return magnetisation_total
121
122 def get_energy():
123     """
124     :return: (float) Total energy through the Hamiltonian
125     """
126     sum1 = 0
127     sum2 = 0
128     for i in range(N):
129         for j in range(number_of_neighbours):
130             sum1 += s[i]*s[neighbours_dictionary[i][j]]
131         if H != 0:
132             sum2 = get_magnetisation()
133     total_energy = (-J*sum1 - mu*H*sum2)/2
134     return total_energy
135
136 def get_average_magnetisation():
137     """
138     :return: (float) Magnetisation per site
139     """
140     return abs(get_magnetisation())/N
141
142 def get_autocovariance(M_list, tau):
143     """
144     :param M_list: (list) holding average magnetisation for each sweep
145     neglected pre-thermalised samples
146     :param tau: (int) time lag which is the input to the autocovariance
147     formula
148     :return: (float) autocovariance for the time lag tau
149     """
150     mean = np.average(M_list)
151     if len(M_list)-tau<=0:
152         print(f'The length of M_list in auto_covariance is {len(M_list)} and
153         of tau {tau}')
154     autocovariance_list = [(M_list[t] - mean)*(M_list[t+tau] - mean) for t in
155     range(len(M_list) - tau)]
156     return np.average(autocovariance_list)
157
158 def get_autocorrelation(M_list, tau):
159     """
160     :param M_list: (list) holding average magnetisation for each sweep
161     neglected pre-thermalised samples
162     :param tau: (int) time lag which is the input to the autocovariance
163     formula
164     :return: (float) autocorrelation for the time lag tau
165     """
166     A_0 = get_autocovariance(M_list, 0)

```



```

161 A_tau = get_autocovariance(M_list, tau)
162 return A_tau/A_0
163
164 def get_target_value_index(autocorrelation_list, target_value):
165     """
166     When the autocorrelation falls below 1/e return the index
167     :param autocorrelation_list: (list) autocorrelation for different tau
168     values
169     :param target_value: (float) 1/e of the initial autocorrelation value
170     :return: (int) Index of tau that gives 1/e of initial autocorrelation
171     """
172     target_index = -1
173     for i in range(len(autocorrelation_list)):
174         if autocorrelation_list[i] < target_value:
175             target_index = i
176             break
177     else:
178         continue
179     return target_index
180
181 def simulation():
182     """
183     Evolves the systems with nsweeps and records average magnetisation for
184     every state. The the simulation
185     finds the tau that makes autocorrelation fall to 1/e called the constant
186     of decorrelation and plots
187     autocorrelation vs tau
188     :return: (void)
189     """
190     avg_magnetisation_list = []
191     autocorrelation_list = []
192     thermalisation_sweeps = 5000
193     sample_every = 50
194
195     for t in range(nsweeps):
196         metropolis()
197         if t < thermalisation_sweeps:
198             continue
199         else:
200             if t % sample_every == 0:
201                 avg_magnetisation_list.append(get_average_magnetisation())
202
203     tau_list = np.arange(0, 50, 1)
204
205     for tau in tau_list:
206         autocorrelation_list.append(get_autocorrelation(
207             avg_magnetisation_list, tau))
208
209     target_value = 1/np.e
210     index = get_target_value_index(autocorrelation_list, target_value)
211     target_tau = tau_list[index]
212
213     plt.plot(tau_list, autocorrelation_list, color = 'k')
214     plt.axhline(y = target_value, linestyle = '--', color = 'r')
215     plt.axvline(x = target_tau, linestyle = '--', color='b')
216     plt.title(f'Autocorrelation \u03b1(\u03c4) vs \u03c4\nTemperature: {T}K,
217             L = {L}')
218     plt.xlabel('\u03c4')
219     plt.ylabel('Autocorrelation \u03b1(\u03c4)')
220     plt.legend(['\u03b1(\u03c4)', '1/e', f'\u03c4 at 1/e = {target_tau}'])
221     plt.grid(True)
222     # plt.savefig(f'autocorrelation_vs_tau_{T}.pdf', bbox = 'tight')
223     plt.show()
224
225 simulation()

```

Code file: correlation_table.py

```

1 from random import uniform, randint
2 import random
3 from math import exp
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import pandas as pd
7 from scipy.optimize import curve_fit
8 import time
9
10 # Global variables
11 J = 1 # Exchange energy
12 H = 0 # Applied magnetic field strength
13 mu = 1 # Magnetic moment
14 M = 0 # Total magnetisation
15 number_of_neighbours = 4
16 random_seed = 10 # Fix random seed in spin initialization for reproducability
17 hot_start = True # Initialize with hot start
18 nsweeps = 75000 # Number of sweeps
19
20 def get_spin_list(is_hot, N):
21     """
22     :param is_hot: (bool) If is_hot is true make hot start otherwise make
23     cold start
24     :param N: (int) N = L x L, total number of sites
25     :return: (list) containing +/- 1 representing the spins on the lattice
26     """
27     random.seed(random_seed)
28     # A hot start means start with +/- 1 with a 50-50 chance on each site
29     if is_hot:
30         s_local = []
31         for i in range(N):
32             rdm_num = uniform(0, 1)
33             if rdm_num > 0.5:
34                 s_local.append(1)
35             else:
36                 s_local.append(-1)
37     # A cold start means start with 1s on all sites
38     else:
39         s_local = [1] * N
40     return s_local
41
42 def get_neighbours_index(L):
43     """
44     :param L: (int) Dimension of the lattice
45     :return: (dict) containing as keys the index of the site on the lattice
46     and as values a list containing the indexes
47     of its neighbours
48     """
49     neighbours_dict = {}
50     N = L**2
51     for i in range(N):
52         # store index of neighbours in the values for each node (key, i) in
53         the lattice
54         # in the form left, right, top, bottom with periodic boundary
55         conditions
56         if i % L == 0:
57             left = i + L - 1
58         else:
59             left = i - 1
60         if (i + 1) % L == 0:
61             right = i - L + 1
62         else:
63             right = i + 1
64         if i - L < 0:
65             top = i - L + N
66         else:
67             top = i - L
68         if i + L >= N:

```

```

65         bottom = i + L - N
66     else:
67         bottom = i + L
68
69     neighbours_dict[i] = [left, right, top, bottom]
70
71     return neighbours_dict
72
73 def get_energy_difference(s, neighbours_dictionary, index_to_flip):
74     """
75     :param s: (list) spin list of the lattice
76     :param neighbours_dictionary: (dict) holds indexes for each site's
77         neighbours
78     :param index_to_flip: (int) the site index to consider flipping its spin
79     :return: (float) Total energy change of changing that site's spin
80     """
81     sum_of_neighbours = 0
82     for neighbour_index in neighbours_dictionary[index_to_flip]:
83         sum_of_neighbours += s[neighbour_index]
84     total_change = 2*s[index_to_flip]*sum_of_neighbours
85     return total_change
86
87 def metropolis(s, neighbours_dictionary, N, dE_4, dE_8):
88     """
89     The metropolis algorithm as a markov chain monte carlo simulation
90     algorithm that modifies the spin state of the
91     lattice and gives a new state by choosing N (= L x L) sites at random and
92     checking through the energy if it will
93     flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
94     spin will be flipped and the numbers 4, 8
95     represent the corresponding change in energy so that we don't calculate
96     many times an exponential term
97     :param s: (list) spin list
98     :param neighbours_dictionary: (dict) holds indexes for neighbours
99     :param N: (int) total number of sites
100     :param dE_4: (float) Probability for energy change of +4
101     :param dE_8: (float) Probability for energy change of +8
102     :return: (void) does not return anything but changes the state of s
103     """
104     for i in range(N):
105         site_index = randint(0,N-1)
106         dE = get_energy_difference(s, neighbours_dictionary, site_index)
107         rdm_num = uniform(0, 1)
108         if dE <= 0:
109             s[site_index] *= -1
110         elif dE == 4:
111             if dE_4 > rdm_num:
112                 s[site_index] *= -1
113             else:
114                 continue
115         elif dE == 8:
116             if dE_8 > rdm_num:
117                 s[site_index] *= -1
118             else:
119                 continue
120
121 def get_magnetisation(s, N):
122     """
123     :param s: (list) spin list
124     :param N: (int) total number of sites
125     :return: (float) Total magnetisation
126     """
127     magnetisation_total = 0
128     for i in range(N):
129         magnetisation_total += s[i]
130     return magnetisation_total
131
132 def get_energy(N, s, neighbours_dictionary):

```

```

128 """
129 :param N: (int) total number of sites
130 :param s: (list) spin list
131 :param neighbours_dictionary: (dict) holds indexes for neighbours
132 :return: (float) Total energy through the Hamiltonian
133 """
134 sum1 = 0
135 sum2 = 0
136 for i in range(N):
137     for j in range(number_of_neighbours):
138         sum1 += s[i]*s[neighbours_dictionary[i][j]]
139         if H != 0:
140             sum2 = get_magnetisation(s, N)
141 total_energy = (-J*sum1 - mu*H*sum2)/2
142 return total_energy
143
144 def get_average_magnetisation(s, N):
145 """
146 :param s: (list) spin list
147 :param N: (int) total number of sites
148 :return: (float) Magnetisation per site
149 """
150 return abs(get_magnetisation(s, N))/N
151
152 def get_autocovariance(M_list, tau):
153 """
154 :param M_list: (list) holding average magnetisation for each sweep
155 :param tau: (int) time lag which is the input to the autocovariance
156 :return: (float) autocovariance for the time lag tau
157 """
158 mean = np.average(M_list)
159 autocovariance_list = [(M_list[t] - mean)*(M_list[t+tau] - mean) for t in
160 range(len(M_list) - tau)]
161 return np.average(autocovariance_list)
162
163 def get_autocorrelation(M_list, tau):
164 """
165 :param M_list: (list) holding average magnetisation for each sweep
166 :param tau: (int) time lag which is the input to the autocovariance
167 :return: (float) autocorrelation for the time lag tau
168 """
169 A_0 = get_autocovariance(M_list, 0)
170 A_tau = get_autocovariance(M_list, tau)
171 return A_tau/A_0
172
173 def get_target_value_index(autocorrelation_list, target_value):
174 """
175 :param autocorrelation_list: (list) autocorrelation for different tau
176 :param target_value: (float) autocorrelation initial value * 1/e
177 :return: (int) index for target tau
178 """
179 target_index = -1
180 for i in range(len(autocorrelation_list)):
181     if autocorrelation_list[i] < target_value:
182         target_index = i
183         break
184     else:
185         continue
186 return target_index
187
188 def get_target_tau(s, L, dE_4, dE_8, neighbours_dictionary):
189 """
190 :param s: (list) spin list

```

```

190 :param L: (int) lattice dimension
191 :param dE_4: (float) Probability for energy change of +4
192 :param dE_8: (float) Probability for energy change of +8
193 :param neighbours_dictionary: (dict) holds indexes for neighbours
194 :return: (int) tau that makes autocorrelation fall to 1/e
195 """
196 N = L**2
197 avg_magnetisation_list = []
198 autocorrelation_list = []
199 tau_list = np.arange(0, 2500, 1)
200 thermalisation_sweeps = 20000
201 sample_every = 1
202
203 for sweep in range(nsweeps):
204
205     metropolis(s, neighbours_dictionary, N, dE_4, dE_8)
206
207     if sweep < thermalisation_sweeps:
208         continue
209     else:
210         if sweep % sample_every == 0:
211             avg_magnetisation_list.append(get_average_magnetisation(s, N)
212 )
213         else:
214             continue
215
216 for tau in tau_list:
217     autocorrelation_list.append(get_autocorrelation(
218         avg_magnetisation_list, tau))
219
220 target_value = 1/np.e
221 index = get_target_value_index(autocorrelation_list, target_value)
222 target_tau = tau_list[index]
223
224 return target_tau
225
226 def simulation():
227     """
228     For different values of L and T the simulation finds tau that makes
229     autocorrelation fall to 1/e.
230     It stores these data in a dataframe and prints an html table of it
231     :return: (void)
232     """
233     tmp_data_list = []
234     T_val = [2.3]
235     T_values = [round(T_val[i], 3) for i in range(len(T_val))]
236     L_values = [10, 20, 30, 40, 50]
237     print(f'Total number of samples: {len(L_values)*len(T_values)}')
238
239     with open('correlation_table.txt', 'w') as file:
240         for L in L_values:
241
242             N = L**2
243             s = get_spin_list(hot_start, N)
244             neighbours_dictionary = get_neighbours_index(L)
245
246             for T in T_values:
247
248                 start = time.process_time()
249                 b = 1 / T # Constant: 1 / temperature
250                 dE_4 = exp(-4 * b) # Probability for energy change of +4
251                 dE_8 = exp(-8 * b) # Probability for energy change of +8
252                 target_tau = get_target_tau(s, L, dE_4, dE_8,
253                     neighbours_dictionary)
254                 # tmp_data_list.append([N, T, target_tau])
255                 time_for_sample = time.process_time() - start
256                 print(f'[L = {L}, T = {T}, tau = {target_tau}] --> Time for
257 sample: {time_for_sample/60:.1f} minutes')

```

```

253         file.write(f'{L},{T},{target_tau}\n')
254
255 def function_to_fit(L_array, z, a, b):
256     """
257     :param L_array: (nparray) Contains the L values tried
258     :param z: (float) z exponent in critical slowing down equation
259     :param a: (float) constant of equation
260     :param b: (float) constant of equation
261     :return: (nparray) The function to fit by finding z, a, b
262     """
263     return a*(L_array**z)+b
264
265 def critical_slowing_down_fit():
266     """
267     Tries to fit the data of tau vs L to find the z exponent in the critical
268     slowing down relationship.
269     It also makes a plot of the data and the fit and prints z
270     :return: (void)
271     """
272     with open('correlation_table.txt', 'r') as file:
273
274         L_list = []
275         T_list = []
276         tau_list = []
277
278         line = file.readline()
279         while line:
280             row = line.strip().split(',')
281             L_list.append(int(row[0]))
282             T_list.append(float(row[1]))
283             tau_list.append(float(row[2]))
284             line = file.readline()
285
286         L_list_df = pd.DataFrame(L_list, columns = ['L'])
287         T_list_df = pd.DataFrame(T_list, columns = ['Temperature'])
288         tau_list_df = pd.DataFrame(tau_list, columns = ['tau'])
289
290         df = pd.concat([L_list_df, T_list_df, tau_list_df], axis = 1)
291         df.to_html('L_T_tau2.html')
292         print(df)
293
294         tau_array = np.array(tau_list)
295         L_array = np.array(L_list)
296
297         p0 = [2.1, 0.2, 1]
298         param_fit, cov = curve_fit(function_to_fit, L_array, tau_array, p0 =
299         p0)
300         print(f'z was found to be: {param_fit[0]}, with standard deviation: {
301         np.sqrt(np.diag(cov)[0])}')
302         z_fit, a_fit, b_fit = param_fit[0], param_fit[1], param_fit[2]
303         tau_fit = a_fit*(L_array**z_fit) + b_fit
304
305         plt.scatter(L_array, tau_array, marker = '+', color = 'r', s = 100)
306         plt.plot(L_array, tau_fit, color = 'k')
307         plt.legend([f'Fit: z = {z_fit:.3f}', 'Simulation'])
308         plt.title('Critical slowing down  $\tau$  vs L\nFit:  $\tau \sim L^z$ \nsim  $L^z$ ')
309         plt.xlabel('L')
310         plt.ylabel('tau')
311         plt.grid(True)
312         # plt.savefig('critical_slowing_down.pdf', bbox = 'tight')
313         plt.show()
314
315 critical_slowing_down_fit()

```

Code file: domains.py

```

1 from random import uniform, randint

```

```

2 import random
3 from math import exp
4 import matplotlib.pyplot as plt
5 import matplotlib.animation as anim
6 import numpy as np
7 import time
8 from collections import Counter
9 from queue import Queue
10
11 # Global variables
12 J = 1 # Exchange energy
13 H = 0
14 mu = 1 # Magnetic moment
15 number_of_neighbours = 4
16 random_seed = 13 # Fix random seed in spin initialization for reproducability
17 hot_start = True # Initialize with hot start or not
18 nsweeps = 10000 # Number of sweeps
19
20 def get_spin_list(is_hot, N):
21     """
22     :param is_hot: (bool) If is_hot is true make hot start otherwise make
23     cold start
24     :param N: (int) N = L x L, total number of sites
25     :return: (list) containing +/- 1 representing the spins on the lattice
26     """
27     random.seed(random_seed)
28     # A hot start means start with +/- 1 with a 50-50 chance on each site
29     if is_hot:
30         s_local = []
31         for i in range(N):
32             rdm_num = uniform(0, 1)
33             if rdm_num > 0.5:
34                 s_local.append(1)
35             else:
36                 s_local.append(-1)
37     # A cold start means start with 1s on all sites
38     else:
39         s_local = [1] * N
40     return s_local
41
42 def get_neighbours_index(N):
43     """
44     :param N: (int) N = L x L, total number of sites
45     :return: (dict) containing as keys the index of the site on the lattice
46     and as values a list containing the indexes
47     of its neighbours
48     """
49     neighbours_dict = {}
50     L = int(N**(1/2))
51     for i in range(N):
52         # store index of neighbours in the values for each node (key, i) in
53         the lattice
54         # in the form left, right, top, bottom with periodic boundary
55         conditions
56         if i % L == 0:
57             left = i + L - 1
58         else:
59             left = i - 1
60         if (i + 1) % L == 0:
61             right = i - L + 1
62         else:
63             right = i + 1
64         if i - L < 0:
65             top = i - L + N
66         else:
67             top = i - L
68         if i + L >= N:
69             bottom = i + L - N

```

```

66         else:
67             bottom = i + L
68
69             neighbours_dict[i] = [left, right, top, bottom]
70
71         return neighbours_dict
72
73 def get_energy_difference(index_to_flip, s, neighbours_dictionary):
74     """
75     :param index_to_flip: (int) the site index to consider flipping its spin
76     :param s: (list) spin list of the lattice
77     :param neighbours_dictionary: (dict) holds indexes for each site's
78         neighbours
79     :return: (float) Total energy change of changing that site's spin
80     """
81     sum_of_neighbours = 0
82     for neighbour_index in neighbours_dictionary[index_to_flip]:
83         sum_of_neighbours += s[neighbour_index]
84     total_change = 2*s[index_to_flip]*sum_of_neighbours + 2*s[index_to_flip]*
85         mu*H
86     return total_change
87
88 def metropolis(N, s, neighbours_dictionary, b):
89     """
90     The metropolis algorithm as a markov chain monte carlo simulation
91     algorithm that modifies the spin state of the
92     lattice and gives a new state by choosing N (= L x L) sites at random and
93     checking through the energy if it will
94     flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
95     spin will be flipped and the numbers 4, 8
96     represent the corresponding change in energy so that we don't calculate
97     many times an exponential term
98     :param b: (float) 1/Temperature
99     :param N: (int) total number of sites
100     :param s: (list) spin list
101     :param neighbours_dictionary: (dict) holds indexes for neighbours
102     :return: (void) does not return anything but changes the state of s
103     """
104     for i in range(N):
105         site_index = randint(0, N-1)
106         dE = get_energy_difference(site_index, s, neighbours_dictionary)
107         val = exp(-dE*b)
108         rdm_num = uniform(0, 1)
109         if dE <= 0:
110             s[site_index] *= -1
111         elif val > rdm_num:
112             s[site_index] *= -1
113         else:
114             continue
115
116 def get_magnetisation(N, s):
117     """
118     :param N: (int) total number of sites
119     :param s: (list) spin list
120     :return: (float) Total magnetisation
121     """
122     magnetisation_total = 0
123     for i in range(N):
124         magnetisation_total += s[i]
125     return magnetisation_total
126
127 def get_energy(N, s, neighbours_dictionary):
128     """
129     :param N: (int) total number of sites
130     :param s: (list) spin list
131     :param neighbours_dictionary: (dict) holds indexes for neighbours
132     :return: (float) Total energy through the Hamiltonian
133     """

```



```

128     sum1 = 0
129     sum2 = 0
130     for i in range(N):
131         for j in range(number_of_neighbours):
132             sum1 += s[i]*s[neighbours_dictionary[i][j]]
133             if H != 0:
134                 sum2 = get_magnetisation(N, s)
135     total_energy = (-J*sum1 - mu*H*sum2)/2
136     return total_energy
137
138 def get_average_energy(N, s, neighbours_dictionary):
139     return get_energy(N, s, neighbours_dictionary)/(2*N)
140
141 def get_average_magnetisation(N, s):
142     """
143     :param N: (int) total number of sites
144     :param s: (list) spin list
145     :return: (float) Magnetisation per site
146     """
147     return get_magnetisation(N, s)/N
148
149 def spin_configuration_2D(s, L):
150
151     x, y = 0, L-1
152     s_2D = [[0 for i in range(L)] for j in range(L)]
153     for idx in range(len(s)):
154
155         if (idx + 1) % L == 0:
156             s_2D[x][y] = s[idx]
157             x = 0
158             y -= 1
159         else:
160             s_2D[x][y] = s[idx]
161             x += 1
162     return s_2D
163
164 def draw_lattice(s, L, T, avg_clust_size):
165
166     s_2D = []
167
168     for i in range(L):
169         s_2D.append(s[i*L:(i+1)*L])
170
171     s_2D_array = np.array(s_2D)
172     plt.imshow(s_2D_array, cmap = 'binary')
173     plt.title(f'L = {L}, T = {T:.2f}, Average cluster size = {avg_clust_size:.2f}')
174     plt.show()
175
176 def bfs(s, index_to_start, neighbours_dictionary):
177     """
178     :param s: (list) lattice spin list
179     :param index_to_start: (int) index to start calculating a cluster from
180     :param neighbours_dictionary: (dictionary) holds indices for the site's
181     neighbours
182     :return: (int, list) cluster size, indices of sites that are now saved in
183     a calculated cluster
184     """
185     cluster_type = s[index_to_start]
186     seen = [False] * len(s)
187     to_explore = Queue()
188     to_explore.put(index_to_start)
189     seen[index_to_start] = True
190     cluster_size = 1
191     cluster_indices = [index_to_start]
192
193     while not to_explore.empty():
194         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
232     temperature for different L dimensions of the
233     lattice. It stores the data in a pandas dataframe and makes a plot of all
234     L in the same average magnetisation vs
235     temperature graph
236     :return: (void)
237     """
238     # Temperature values in Kelvin
239     T_extra = list(np.linspace(2.2, 2.5, 11))
240     T_values = list(np.linspace(2, 2.8, 31))
241     T_values.extend(T_extra)
242     T_values.sort()
243     # Dimension values
244     L_values = [12, 24, 32]
245     # Independent sampling
246     thermalisation_sweeps = 5000
247     sample_every = 50
248
249     with open('domains.txt', 'w') as file:
250
251         for L in L_values:
252
253             N = L**2
254             s = get_spin_list(hot_start, N)
255             neighbours_dictionary = get_neighbours_index(N)
256
257             for T in T_values:

```

```

257         b = 1/T # Constant: 1 / temperature
258         cluster_size_list = []
259         start = time.process_time()
260
261         for i in range(nsweeps):
262             metropolis(N, s, neighbours_dictionary, b)
263             if i < thermalisation_sweeps:
264                 continue
265             elif i % sample_every == 0:
266                 cluster_size_list.append(get_average_cluster_size(s,
neighbours_dictionary))
267
268             cluster_std = np.std(cluster_size_list)
269             cluster_avg = np.average(cluster_size_list)
270
271             file.write(f'{L},{T},{cluster_avg},{cluster_std}\n')
272             time_for_sample = time.process_time() - start
273             print(f'L = {L}, T = {T:.2f}, Cluster size = {cluster_avg:.2f
}, Std = {cluster_std:.2f} --> Time for sample = {time_for_sample:.2f
seconds}')
274
275 def domains_plotter():
276
277     with open('domains.txt', 'r') as file:
278
279         L_list = []
280         T_list = []
281         cluster_values = []
282         cluster_error = []
283
284         line = file.readline()
285         while line:
286             row = line.strip().split(',')
287             L_list.append(int(row[0]))
288             T_list.append(float(row[1]))
289             cluster_values.append(float(row[2]))
290             cluster_error.append(float(row[3]))
291             line = file.readline()
292
293             L_values = list(Counter(L_list).keys())
294             T_values = list(Counter(T_list).keys())
295             T_len = len(T_values)
296             L_str_values = ['$T_c$']
297             L_str_values.extend(['L = ' + str(L_values[i]) for i in range(len(
L_values))])
298
299             for i in range(len(L_values)-1):
300                 i += 1
301                 plt.errorbar(T_values, cluster_values[i*T_len:(i+1)*T_len], yerr
= cluster_error[i*T_len:(i+1)*T_len],
302                             marker = '+', label = f'L = {L_values[i]}',
errorevery = 5, ecolor = 'r', capsize = 2)
303
304                 plt.axvline(x=2 / np.log(1 + np.sqrt(2)), color='k', linestyle='--',
label='$T_c$')
305                 plt.legend()
306                 plt.xlabel('T (J/$k_B$)')
307                 plt.ylabel('Average Cluster size')
308                 plt.title('Average Cluster size vs Temperature')
309                 plt.grid()
310                 plt.savefig('domains1.pdf', bbox = 'tight')
311                 plt.show()
312
313 #simulation()
314 domains_plotter()

```

Code file: heat_capacity.py

```

1 from random import uniform, randint
2 import random
3 from math import exp
4 import numpy as np
5 from bootstrap import bootstrap
6 from heat_capacity_plotter import heat_capacity_plotter
7 import time
8 import pandas as pd
9
10 # Global variables
11 J = 1 # Exchange energy
12 H = 0 # Applied magnetic field strength
13 mu = 1 # Magnetic moment
14 M = 0 # Total magnetisation
15 random_seed = 13
16 number_of_neighbours = 4
17 hot_start = False # Initialize with hot start
18 nsweeps = 60000 # Number of sweeps
19
20 def get_spin_list(is_hot, N):
21     """
22     :param is_hot: (bool) If is_hot is true make hot start otherwise make
23     cold start
24     :param N: (int) N = L x L, total number of sites
25     :return: (list) containing +/- 1 representing the spins on the lattice
26     """
27     random.seed(random_seed)
28     # A hot start means start with +/- 1 with a 50-50 chance on each site
29     if is_hot:
30         s_local = []
31         for i in range(N):
32             rdm_num = uniform(0, 1)
33             if rdm_num > 0.5:
34                 s_local.append(1)
35             else:
36                 s_local.append(-1)
37     # A cold start means start with 1s on all sites
38     else:
39         s_local = [1] * N
40     return s_local
41
42 def get_neighbours_index(N):
43     """
44     :param N: (int) N = L x L, total number of sites
45     :return: (dict) containing as keys the index of the site on the lattice
46     and as values a list containing the indexes
47     of its neighbours
48     """
49     neighbours_dict = {}
50     L = int(N**(1/2))
51     for i in range(N):
52         # store index of neighbours in the values for each node (key, i) in
53         the lattice
54         # in the form left, right, top, bottom with periodic boundary
55         conditions
56         if i % L == 0:
57             left = i + L - 1
58         else:
59             left = i - 1
60         if (i + 1) % L == 0:
61             right = i - L + 1
62         else:
63             right = i + 1
64         if i - L < 0:
65             top = i - L + N
66         else:
67             top = i - L
68         if i + L >= N:

```

```

65         bottom = i + L - N
66     else:
67         bottom = i + L
68
69     neighbours_dict[i] = [left, right, top, bottom]
70
71     return neighbours_dict
72
73 def get_energy_difference(index_to_flip, s, neighbours_dictionary):
74     """
75     :param index_to_flip: (int) the site index to consider flipping its spin
76     :param s: (list) spin list of the lattice
77     :param neighbours_dictionary: (dict) holds indexes for each site's
78         neighbours
79     :return: (float) Total energy change of changing that site's spin
80     """
81     sum_of_neighbours = 0
82     for neighbour_index in neighbours_dictionary[index_to_flip]:
83         sum_of_neighbours += s[neighbour_index]
84     total_change = 2*s[index_to_flip]*sum_of_neighbours
85     return total_change
86
87 def metropolis(dE_4, dE_8, N, s, neighbours_dictionary):
88     """
89     The metropolis algorithm as a markov chain monte carlo simulation
90     algorithm that modifies the spin state of the
91     lattice and gives a new state by choosing N (= L x L) sites at random and
92     checking through the energy if it will
93     flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
94     spin will be flipped and the numbers 4, 8
95     represent the corresponding change in energy so that we don't calculate
96     many times an exponential term
97     :param dE_4: (float) Probability for energy change of +4
98     :param dE_8: (float) Probability for energy change of +8
99     :param N: (int) total number of sites
100    :param s: (list) spin list
101    :param neighbours_dictionary: (dict) holds indexes for neighbours
102    :return: (void) does not return anything but changes the state of s
103    """
104    for i in range(N):
105        site_index = randint(0, N-1)
106        dE = get_energy_difference(site_index, s, neighbours_dictionary)
107        rdm_num = uniform(0, 1)
108        if dE <= 0:
109            s[site_index] *= -1
110        elif dE == 4:
111            if dE_4 > rdm_num:
112                s[site_index] *= -1
113            else:
114                continue
115        elif dE == 8:
116            if dE_8 > rdm_num:
117                s[site_index] *= -1
118            else:
119                continue
120
121 def get_magnetisation(N, s):
122     """
123     :param N: (int) total number of sites
124     :param s: (list) spin list
125     :return: (float) Total magnetisation
126     """
127     magnetisation_total = 0
128     for i in range(N):
129         magnetisation_total += s[i]
130     return magnetisation_total
131
132 def get_average_energy(N, s, neighbours_dictionary):

```

```

128 """
129 :param N: (int) total number of sites
130 :param s: (list) spin list
131 :param neighbours_dictionary: (dict) holds indexes for neighbours
132 :return: (float) Total energy through the Hamiltonian
133 """
134 sum1 = 0
135 sum2 = 0
136 for i in range(N):
137     for j in range(number_of_neighbours):
138         sum1 += s[i]*s[neighbours_dictionary[i][j]]
139         if H != 0:
140             sum2 = get_magnetisation(N, s)
141 total_energy = (-J*sum1 - mu*H*sum2)/2
142 return total_energy/2*N
143
144 def get_energy(N, s, neighbours_dictionary):
145 """
146 :param N: (int) total number of sites
147 :param s: (list) spin list
148 :param neighbours_dictionary: (dict) holds indexes for neighbours
149 :return: (float) Total energy through the Hamiltonian
150 """
151 sum1 = 0
152 sum2 = 0
153 for i in range(N):
154     for j in range(number_of_neighbours):
155         sum1 += s[i]*s[neighbours_dictionary[i][j]]
156         if H != 0:
157             sum2 = get_magnetisation(N, s)
158 total_energy = (-J*sum1 - mu*H*sum2)/2
159 return total_energy
160
161 def get_average_magnetisation(N, s):
162 """
163 :param N: (int) total number of sites
164 :param s: (list) spin list
165 :return: (float) Magnetisation per site
166 """
167 return abs(get_magnetisation(N, s))/N
168
169 def get_autocovariance(M_list, tau):
170 """
171 :param M_list: (list) holding average magnetisation for each sweep
172               neglected pre-thermalised samples
173 :param tau: (int) time lag which is the input to the autocovariance
174             formula
175 :return: (float) autocovariance for the time lag tau
176 """
177 mean = np.average(M_list)
178 autocovariance_list = [(M_list[t] - mean)*(M_list[t+tau] - mean) for t in
179                        range(len(M_list) - tau)]
180 return np.average(autocovariance_list)
181
182 def get_autocorrelation(M_list, tau):
183 """
184 :param M_list: (list) holding average magnetisation for each sweep
185               neglected pre-thermalised samples
186 :param tau: (int) time lag which is the input to the autocovariance
187             formula
188 :return: (float) autocorrelation for the time lag tau
189 """
190 A_0 = get_autocovariance(M_list, 0)
191 A_tau = get_autocovariance(M_list, tau)
192 return A_tau/A_0
193
194 def get_target_value_index(autocorrelation_list, target_value):
195 """

```

```

191 :param autocorrelation_list: (list) autocorrelation for different tau
    values
192 :param target_value: (float) autocorrelation initial value * 1/e
193 :return: (int) index for target tau
194 """
195 target_index = -1
196 for i in range(len(autocorrelation_list)):
197     if autocorrelation_list[i] < target_value:
198         target_index = i
199         break
200     else:
201         continue
202 return target_index
203
204 def get_target_tau(avg_mag_list):
205     """
206     :param avg_mag_list: (list) holds total magnetisation for each sweep
    state
207     :return: (int) tau that makes autocorrelation fall to 1/e
208     """
209     autocorrelation_list = []
210     tau_list = np.arange(0, 50, 1)
211
212     for tau in tau_list:
213         autocorrelation_list.append(get_autocorrelation(avg_mag_list, tau))
214
215     target_value = 1/np.e
216     index = get_target_value_index(autocorrelation_list, target_value)
217     target_tau = tau_list[index]
218
219     return target_tau
220
221 def simulation():
222     """
223     Generates data for C vs T for different values of L along with standard
    deviation for C using bootstrap.
224     The data are written in a txt file and plotted using another file called
    heat_capacity_plotter.py. This
225     function also finds the Tc by taking the T input that gives maximum C and
    prints a table of L vs Tc
226     :return: (void)
227     """
228     T_val = np.linspace(2, 3, 100)
229     T_values = [round(T_val[i], 3) for i in range(len(T_val))]
230     L_values = [64]
231     print(f'Total samples to calculate: {len(T_values)*len(L_values)}')
232     n_bins = 100
233     Tc_list = []
234     thermalisation_sweeps = 10000
235     sample_every = 50
236
237     with open('heat_capacity_data1.txt', 'w') as file:
238         for L in L_values:
239
240             C_list, sigma_C_list = [], []
241             N = L ** 2
242             s = get_spin_list(hot_start, N)
243             neighbours_dictionary = get_neighbours_index(N)
244
245             for T in T_values:
246
247                 start = time.process_time()
248                 avg_mag_list = []
249                 energy_list = []
250                 b = 1 / T # Constant: 1 / temperature
251                 dE_4 = exp(-4 * b) # Probability for energy change of +4
252                 dE_8 = exp(-8 * b) # Probability for energy change of +8
253

```

```

254         for sweep in range(nsweps):
255
256             metropolis(dE_4, dE_8, N, s, neighbours_dictionary)
257
258             if sweep < thermalisation_sweeps:
259                 continue
260             else:
261                 if sweep % sample_every == 0:
262                     avg_mag_list.append(get_average_magnetisation(N,
263 s))
264                     energy_list.append(get_energy(N, s,
265 neighbours_dictionary))
266
267                     target_tau = get_target_tau(avg_mag_list)
268                     C, sigma_C = bootstrap(energy_list, n_bins, T, target_tau)
269                     C_list.append(C)
270                     sigma_C_list.append(sigma_C)
271                     time_for_sample = time.process_time() - start
272
273                     file.write(f'{L},{T},{C},{sigma_C}\n')
274                     print(f'[L = {L}, T = {T}, C = {C:.2f}, sigma = {sigma_C:.2f}
275 ], tau = {target_tau}]', f'--> Time for sample: {time_for_sample/60:.1f}
276 minutes')
277
278                     index_for_Tc = C_list.index(max(C_list))
279                     Tc_list.append(T_values[index_for_Tc])
280
281                     L_df = pd.DataFrame(L_values, columns = ['L'])
282                     Tc_df = pd.DataFrame(Tc_list, columns = ['Tc'])
283                     Tc_data = pd.concat([L_df, Tc_df], axis = 1)
284                     print(Tc_data)
285                     Tc_data.to_html('Tc_vs_L_table1.html')
286
287 simulation()
288 heat_capacity_plotter()

```

Code file: heat_capacity_plotter.py

```

1 import matplotlib.pyplot as plt
2 from collections import Counter
3 import numpy as np
4 from scipy.optimize import curve_fit
5 import pandas as pd
6
7 def heat_capacity_plotter():
8     """
9     Plots C/N vs T for different values of L
10    :return: (void)
11    """
12    with open('heat_capacity_data1.txt', 'r') as file:
13
14        onsager_Tc = 2/np.log(1+np.sqrt(2))
15
16        L_list = []
17        T_list = []
18        C_list = []
19        sigma_C_list = []
20
21        line = file.readline()
22        while line:
23            row = line.strip().split(',')
24            L_list.append(int(row[0]))
25            T_list.append(float(row[1]))
26            C_list.append(float(row[2]))
27            sigma_C_list.append(float(row[3]))
28            line = file.readline()
29
30        L_values = list(Counter(L_list).keys())

```



```

31     L_length = len(L_values)
32     T_length = len(list(Counter(T_list).keys()))
33
34     L_str_values = ['L = ' + str(L_values[i]) for i in range(len(L_values
35 ))]
36     L_str_values.insert(0, 'Onsager\'s  $T_c$ ')
37
38     C_array = np.array(C_list)
39     sigma_C_array = np.array(sigma_C_list)
40     N_values_array = np.array(L_values)**2
41
42     colour_list = ['k', 'b', 'g', 'y', 'm', 'c', 'r', 'navy', 'lightcoral',
43                   'lime']
44
45     for i in range(L_length):
46         plt.errorbar(T_list[i*T_length:(i+1)*T_length], C_array[i*
47 T_length:(i+1)*T_length]/N_values_array[i],
48                     xerr = 0, yerr = sigma_C_array[i*T_length:(i+1)*
49 T_length]/N_values_array[i], ls = '',
50                     marker = '+', ecolor = 'r', capsize = 2, color =
51 colour_list[i], elinewidth = 0.7, errorevery = 30)
52
53     plt.axvline(x=onsager_Tc, color='k', linestyle='--')
54     # plt.yscale('log')
55     plt.legend(L_str_values)
56     plt.title('Specific heat capacity per site vs temperature')
57     plt.xlabel('T (J/$k_B$)')
58     plt.ylabel('C/N')
59     # plt.savefig('heat_cap_vs_tmp2.pdf', bbox = 'tight')
60     plt.show()
61
62     tc_fitter(L_list, T_list, C_list, sigma_C_list)
63
64 def gaussian(x, mu, sigma, amp):
65     return amp*np.exp(-(x-mu)/sigma)**2)
66
67 def tc_fitter(L_values, T_values, C_values, sigma_C_values):
68
69     # L_values = [4, 8, 10, 12, 24, 32, 48]
70     # tc_values = [2.394, 2.354, 2.364, 2.384, 2.293, 2.293, 2.273]
71
72     L_val = list(Counter(L_values).keys())
73     L_len = len(L_val)
74     T_len = len(list(Counter(T_values).keys()))
75
76     tc_list = []
77     tc_error = []
78
79     with open('finite_scaling_data.txt', 'w') as file:
80         for i in range(L_len):
81
82             T_tmp = T_values[i*T_len:(i+1)*T_len]
83             C_tmp = C_values[i*T_len:(i+1)*T_len]
84             sigma_C_tmp = sigma_C_values[i*T_len:(i+1)*T_len]
85             index_of_max = C_tmp.index(max(C_tmp))
86             T = T_tmp[index_of_max - 20:index_of_max + 20]
87             C = C_tmp[index_of_max - 20:index_of_max + 20]
88             C_sigma = sigma_C_tmp[index_of_max - 20:index_of_max + 20]
89             parameters, pcov = curve_fit(gaussian, T, C, sigma = C_sigma,
90 absolute_sigma = True)
91             mu, sigma, amp, sigma_mu = parameters[0], parameters[1],
92 parameters[2], np.sqrt(np.diag(pcov)[0])
93             tc_list.append(mu)
94             tc_error.append(sigma_mu)
95             file.write(f'{L_val[i]**2},{mu},{sigma_mu}\n')
96
97     L_df = pd.DataFrame(L_val, columns = ['L'])

```

```

92 tc_df = pd.DataFrame(tc_list, columns = ['Tc'])
93 tc_error_df = pd.DataFrame(tc_error, columns = ['sigma_Tc'])
94 df = pd.concat([L_df, tc_df, tc_error_df], axis = 1)
95 # df.to_html('L_Tc_error.html')
96 print(df)
97
98
99 def func(L, tc_at_inf, a, nu):
100     return tc_at_inf + a*L**(-1/nu)
101
102 def finite_scaling_fitter():
103
104     with open('finite_scaling_data.txt', 'r') as file:
105
106         L_list = []
107         tc_list = []
108         tc_error = []
109
110         line = file.readline()
111         while line:
112             row = line.strip().split(',')
113             L_list.append(float(row[0]))
114             tc_list.append(float(row[1]))
115             tc_error.append(float(row[2]))
116             line = file.readline()
117
118         p0 = [2.26, 1.0, 2.0]
119
120         parameters, covariance_matrix = curve_fit(func, L_list, tc_list,
121                                                    sigma = tc_error,
122                                                    absolute_sigma = True, p0 = p0)
123         tc_at_inf, a, nu = parameters[0], parameters[1], parameters[2]
124         tc_at_inf_error, a_error, nu_error = np.sqrt(np.diag(
125             covariance_matrix)[0]), \
126                                     np.sqrt(np.diag(
127             covariance_matrix)[1]), \
128                                     np.sqrt(np.diag(
129             covariance_matrix)[2])
130
131         print(f'\nTc at infinity: {tc_at_inf:.3f}, sigma: {tc_at_inf_error:.3f}\na: {a:.3f}, '
132               f'sigma: {a_error:.3f}\nnu: {nu:.3f}, sigma: {nu_error:.3f}')
133
134         plt.plot(L_list, tc_list, color = 'k', linestyle = '--')
135         plt.plot(L_list, tc_at_inf + a*np.array(L_list)**(-1/nu), color = 'r')
136
137         plt.title('Finite-size scaling: $T_c$ vs $N$ Fit: $T_c(N) = T_c(\u221e) + \u03B1(N)^{-\frac{1}{\nu}}$')
138         plt.ylabel('$T_c(N)$')
139         plt.xlabel('$N$')
140         plt.legend(['Simulation', 'Fit'])
141         plt.savefig('finite_size_scaling.pdf', bbox = 'tight')
142         plt.show()
143
144 heat_capacity_plotter()
145 finite_scaling_fitter()

```

Code file: hysteresis.py

```

1 from random import uniform, randint
2 import random
3 from math import exp
4 import matplotlib.pyplot as plt
5 import numpy as np
6 import time
7 from collections import Counter
8 import turtle
9

```

```

10 # Global variables
11 J = 1 # Exchange energy
12 mu = 1 # Magnetic moment
13 number_of_neighbours = 4
14 random_seed = 13 # Fix random seed in spin initialization for reproducibility
15 hot_start = False # Initialize with hot start or not
16 nsweeps = 8000 # Number of sweeps
17
18 def get_spin_list(is_hot, N):
19     """
20     :param is_hot: (bool) If is_hot is true make hot start otherwise make
21     cold start
22     :param N: (int) N = L x L, total number of sites
23     :return: (list) containing +/- 1 representing the spins on the lattice
24     """
25     random.seed(random_seed)
26     # A hot start means start with +/- 1 with a 50-50 chance on each site
27     if is_hot:
28         s_local = []
29         for i in range(N):
30             rdm_num = uniform(0, 1)
31             if rdm_num > 0.5:
32                 s_local.append(1)
33             else:
34                 s_local.append(-1)
35     # A cold start means start with 1s on all sites
36     else:
37         s_local = [1] * N
38     return s_local
39
40 def get_neighbours_index(N):
41     """
42     :param N: (int) N = L x L, total number of sites
43     :return: (dict) containing as keys the index of the site on the lattice
44     and as values a list containing the indexes
45     of its neighbours
46     """
47     neighbours_dict = {}
48     L = int(N**(1/2))
49     for i in range(N):
50         # store index of neighbours in the values for each node (key, i) in
51         # the lattice
52         # in the form left, right, top, bottom with periodic boundary
53         # conditions
54         if i % L == 0:
55             left = i + L - 1
56         else:
57             left = i - 1
58         if (i + 1) % L == 0:
59             right = i - L + 1
60         else:
61             right = i + 1
62         if i - L < 0:
63             top = i - L + N
64         else:
65             top = i - L
66         if i + L >= N:
67             bottom = i + L - N
68         else:
69             bottom = i + L
70
71         neighbours_dict[i] = [left, right, top, bottom]
72     return neighbours_dict
73
74 def get_energy_difference(index_to_flip, s, neighbours_dictionary, H):
75     """
76     :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
88     algorithm that modifies the spin state of the
89     lattice and gives a new state by choosing N (= L x L) sites at random and
90     checking through the energy if it will
91     flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
92     spin will be flipped and the numbers 4, 8
93     represent the corresponding change in energy so that we don't calculate
94     many times an exponential term
95     :param b: (float) 1/Temperature
96     :param H: (float) Applied magnetic field
97     :param N: (int) total number of sites
98     :param s: (list) spin list
99     :param neighbours_dictionary: (dict) holds indexes for neighbours
100     :return: (void) does not return anything but changes the state of s
101     """
102     for i in range(N):
103         site_index = randint(0, N-1)
104         dE = get_energy_difference(site_index, s, neighbours_dictionary, H)
105         val = exp(-dE*b)
106         rdm_num = uniform(0, 1)
107         if dE <= 0:
108             s[site_index] *= -1
109         elif val > rdm_num:
110             s[site_index] *= -1
111         else:
112             continue
113
114 def get_magnetisation(N, s):
115     """
116     :param N: (int) total number of sites
117     :param s: (list) spin list
118     :return: (float) Total magnetisation
119     """
120     magnetisation_total = 0
121     for i in range(N):
122         magnetisation_total += s[i]
123     return magnetisation_total
124
125 def get_energy(N, s, neighbours_dictionary, H):
126     """
127     :param H: (float) Applied magnetic field
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)

```

```

136     total_energy = (-J*sum1 - mu*H*sum2)/2
137     return total_energy
138
139 def get_average_energy(N, s, neighbours_dictionary, H):
140     return get_energy(N, s, neighbours_dictionary, H)/(2*N)
141
142 def get_average_magnetisation(N, s):
143     """
144     :param N: (int) total number of sites
145     :param s: (list) spin list
146     :return: (float) Magnetisation per site
147     """
148     return get_magnetisation(N, s)/N
149
150 def spin_configuration_2D(s, L):
151
152     x, y = 0, L-1
153     s_2D = [[0 for i in range(L)] for j in range(L)]
154     for idx in range(len(s)):
155
156         if (idx + 1) % L == 0:
157             s_2D[x][y] = s[idx]
158             x = 0
159             y -= 1
160         else:
161             s_2D[x][y] = s[idx]
162             x += 1
163     return s_2D
164
165 def draw_lattice(s, L, T, avg_clust_size):
166
167     s_2D = []
168
169     for i in range(L):
170         s_2D.append(s[i*L:(i+1)*L])
171
172     s_2D_array = np.array(s_2D)
173     plt.imshow(s_2D_array, cmap = 'binary')
174     plt.title(f'L = {L}, T = {T:.2f}, Average cluster size = {avg_clust_size:.2f}')
175     plt.show()
176
177 def simulation():
178     """
179     The simulation function gathers data for average magnetisation vs
180     temperature for different L dimensions of the
181     lattice. It stores the data in a pandas dataframe and makes a plot of all
182     L in the same average magnetisation vs
183     temperature graph
184     :return: (void)
185     """
186     # Temperature values in Kelvin
187     T_values = [2.4, 2.6, 3]
188     # Dimension values
189     L_values = [10]
190     # Applied magnetic field values
191     extra_H_neg = list(np.linspace(-0.05128, -0.05385, 5))
192     H_val = list(np.linspace(0, 1.0, 21))
193     H_val.extend(H_val[::-1])
194     H_neg = list(np.linspace(0, -1.0, 21))
195     H_neg.extend(extra_H_neg)
196     H_neg.sort(reverse=True)
197     H_val.extend(H_neg)
198     H_values = [round(H_val[i], 10) for i in range(len(H_val))]
199     tmp_list = [H_values[0]]
200     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,
236 s, neighbours_dictionary, H))
237
238                 mean_average_magnetisation = np.average(
239 average_magnetisation_list)
240                 mean_energy = np.average(energy_per_site_list)
241                 file.write(f'{L},{T},{H},{mean_average_magnetisation},{
242 mean_energy}\n')
243                 time_for_sample = time.process_time() - start
244                 print(f'L = {L}, T = {T}, H = {H}, <m> = {
245 mean_average_magnetisation:.5f}, <E> = {mean_energy:.2f} --> Time for
246 sample = {time_for_sample:.2f} seconds')
247
248 def hysteresis_plotter():
249
250     magnetisation_per_site_list = []
251     energy_per_site = []
252     L_list = []
253     T_list = []
254     H_list = []
255
256     with open('hysteresis.txt', 'r') as file:
257
258         line = file.readline()
259         count = 0
260
261         while line:
262             row = line.strip().split(',')
263             L_list.append(int(row[0]))
264             T_list.append(float(row[1]))
265             magnetisation_per_site_list.append(float(row[3]))
266             energy_per_site.append(float(row[4]))
267             if count == 3:
268                 line = file.readline()

```

```

264         continue
265     else:
266         H_list.append(float(row[2]))
267         if float(row[2]) == 0:
268             count += 1
269         line = file.readline()
270
271     L_values = list(Counter(L_list).keys())
272     T_values = list(Counter(T_list).keys())
273     H_values = H_list
274     H_len = len(H_values)
275     T_str_values = ['T = ' + str(T_values[i]) for i in range(len(T_values
))]
276
277     for i in range(len(T_values)):
278         plt.scatter(H_values, magnetisation_per_site_list[i*H_len:(i+1)*
H_len], marker = '+', s = None)
279         plt.title(f'Hysteresis loop (L = {L_values[0]})\nMagnetisation per
site vs applied field')
280         plt.xlabel('H')
281         plt.ylabel('<m>')
282         plt.legend(T_str_values)
283         plt.grid()
284         # plt.savefig('hysteresis_mag.pdf', bbox = 'tight')
285         plt.show()
286
287     for i in range(len(T_values)):
288         plt.scatter(H_values, energy_per_site[i * H_len:(i + 1) * H_len],
marker = '+', s = None)
289         plt.title(f'Energy per site vs applied field (L = {L_values[0]})')
290         plt.xlabel('H')
291         plt.ylabel('<E>')
292         plt.grid()
293         plt.legend(T_str_values)
294         # plt.savefig('hysteresis_energy.pdf', bbox = 'tight')
295         plt.show()
296
297 #simulation()
298 hysteresis_plotter()

```

Code file: nearest_neighbours.py

```

1 from random import uniform, randint
2 import random
3 from math import exp
4 from nearest_neighbours_plottter import nearest_neighbours_plotter
5 import numpy as np
6 import time
7
8 # Global variables
9 H = 0 # Applied magnetic field strength
10 mu = 1 # Magnetic moment
11 M = 0 # Total magnetisation
12 random_seed = 10 # Fix random seed in spin initialization for reproducibility
13 hot_start = False # Initialize with hot start or not
14 nsweeps = 2000 # Number of sweeps
15
16 def get_spin_list(is_hot, N):
17     """
18     :param is_hot: (bool) If is_hot is true make hot start otherwise make
cold start
19     :param N: (int) N = L x L, total number of sites
20     :return: (list) containing +/- 1 representing the spins on the lattice
21     """
22     random.seed(random_seed)
23     # A hot start means start with +/- 1 with a 50-50 chance on each site
24     if is_hot:
25         s_local = []

```

```

26         for i in range(N):
27             rdm_num = uniform(0, 1)
28             if rdm_num > 0.5:
29                 s_local.append(1)
30             else:
31                 s_local.append(-1)
32         # A cold start means start with 1s on all sites
33     else:
34         s_local = [1] * N
35     return s_local
36
37 def get_neighbours_index(N):
38     """
39     :param N: (int) N = L x L, total number of sites
40     :return: (dict) containing as keys the index of the site on the lattice
41             and as values a list containing the indexes
42             of its neighbours
43     """
44     neighbours_dict = {}
45     L = int(N**(1/2))
46     for i in range(N):
47         # store index of neighbours in the values for each node (key, i) in
48         # the lattice
49         # in the form left, right, top, bottom with periodic boundary
50         # conditions
51         if i % L == 0:
52             left = i + L - 1
53         else:
54             left = i - 1
55         if (i + 1) % L == 0:
56             right = i - L + 1
57         else:
58             right = i + 1
59         if i - L < 0:
60             top = i - L + N
61         else:
62             top = i - L
63         if i + L >= N:
64             bottom = i + L - N
65         else:
66             bottom = i + L
67
68         neighbours_dict[i] = [left, right, top, bottom]
69
70     return neighbours_dict
71
72 def get_nearest_neighbours_index(N):
73     """
74     :param N: (int) Total number of sites
75     :return: (dictionary) Keys - sites indices, values - next nearest
76             neighbours' indices
77     """
78     L = int(N ** (1 / 2)) # Lattice dimension
79     nearest_neighbours_dictionary = {} # Dictionary to return
80
81     for i in range(N):
82         # Up-left neighbour
83         if i == 0: # If the site is index 0
84             upleft = N - 1
85         elif i % L == 0: # If the site is in the first column
86             upleft = i - 1
87         elif i - L < 0: # If the site is in the first row
88             upleft = i - L + N - 1
89         else:
90             upleft = i - L - 1
91
92         # Up-right neighbour
93         if i == L - 1: # If the site is at the top right corner
94             upright = L * (L - 1)

```



```

90         elif (i + 1) % L == 0: # If the site is in the last column
91             upright = i - 2 * L + 1
92         elif i - L < 0: # If the site is in the first row
93             upright = i - L + N + 1
94         else:
95             upright = i - L + 1
96         # Down-left neighbour
97         if i == L * (L - 1): # If the site is at the bottom left corner
98             downleft = L - 1
99         elif i + L >= N: # If the site is in the last row
100             downleft = i + L - N - 1
101         elif i % L == 0: # If the site is in the first column
102             downleft = i + 2 * L - 1
103         else:
104             downleft = i + L - 1
105         # Down-right neighbour
106         if i == N - 1: # If the site is at the down right corner
107             downright = 0
108         elif (i + 1) % L == 0: # If the site is in the last column
109             downright = i + 1
110         elif i + L >= N: # If the site is in the last row
111             downright = i + L - N + 1
112         else:
113             downright = i + L + 1
114
115         nearest_neighbours_dictionary[i] = [upleft, upright, downleft,
116                                           downright]
117
118     return nearest_neighbours_dictionary
119
120 def get_energy_difference(index_to_flip, s, neighbours_dictionary,
121                           nearest_neighbours_dictionary, J):
122     """
123     :param index_to_flip: (int) the site index to consider flipping its spin
124     :param s: (list) spin list of the lattice
125     :param neighbours_dictionary: (dict) holds indexes for each site's
126     neighbours
127     :param nearest_neighbours_dictionary: (dict) holds indexes for each site's
128     nearest neighbours
129     :param J: (float) exchange energy
130     :return: (float) Total energy change of changing that site's spin
131     """
132     sum_of_neighbours = 0
133     sum_of_nearest_neighbours = 0
134     for neighbour_index in neighbours_dictionary[index_to_flip]:
135         sum_of_neighbours += s[neighbour_index]
136     for nearest_neighbour_index in nearest_neighbours_dictionary[
137         index_to_flip]:
138         sum_of_nearest_neighbours += s[nearest_neighbour_index]
139     total_change = 2*s[index_to_flip]*sum_of_neighbours + 2*J*s[index_to_flip]
140     *sum_of_nearest_neighbours
141     return total_change
142
143 def metropolis(N, s, neighbours_dictionary, nearest_neighbours_dictionary, J,
144               b):
145     """
146     :param N: (int) total number of sites
147     :param s: (list) spin list
148     :param neighbours_dictionary: (dict) holds indexes for neighbours
149     :param nearest_neighbours_dictionary: (dict) holds indexes for nearest
150     neighbours
151     :param J: (float) exchange energy for next-nearest neighbours only (J=1
152     for neighbours)
153     :param b: (float) 1/Temperature
154     :return: (void) does not return anything but changes the state of s
155     """
156     for i in range(N):
157         site_index = randint(0, N-1)

```

```

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

```

```

214         time_for_sample = time.process_time() - start
215         file.write(f'{L},{J},{T},{np.average(tmp_magnetisation)}\n')
216         print(f'L = {L}, J = {J}, T = {T:.2f}, <m> = {np.average(
217             tmp_magnetisation):.5f}, Time for sample = {time_for_sample:.2f} seconds')
218 simulation()
219 nearest_neighbours_plotter()

```

Code file: susceptibility.py

```

1 from random import uniform, randint
2 import random
3 from math import exp
4 import numpy as np
5 from bootstrap_susceptibility import bootstrap
6 from susceptibility_plotter import susceptibility_plotter
7 import time
8
9 # Global variables
10 J = 1 # Exchange energy
11 H = 0 # Applied magnetic field strength
12 mu = 1 # Magnetic moment
13 M = 0 # Total magnetisation
14 random_seed = 11
15 number_of_neighbours = 4
16 hot_start = True # Initialize with hot start
17 nsweeps = 10000 # Number of sweeps
18
19 def get_spin_list(is_hot, N):
20     """
21     :param is_hot: (bool) If is_hot is true make hot start otherwise make
22     cold start
23     :param N: (int) N = L x L, total number of sites
24     :return: (list) containing +/- 1 representing the spins on the lattice
25     """
26     random.seed(random_seed)
27     # A hot start means start with +/- 1 with a 50-50 chance on each site
28     if is_hot:
29         s_local = []
30         for i in range(N):
31             rdm_num = uniform(0, 1)
32             if rdm_num > 0.5:
33                 s_local.append(1)
34             else:
35                 s_local.append(-1)
36     # A cold start means start with 1s on all sites
37     else:
38         s_local = [1] * N
39     return s_local
40
41 def get_neighbours_index(N):
42     """
43     :param N: (int) N = L x L, total number of sites
44     :return: (dict) containing as keys the index of the site on the lattice
45     and as values a list containing the indexes
46     of its neighbours
47     """
48     neighbours_dict = {}
49     L = int(N**(1/2))
50     for i in range(N):
51         # store index of neighbours in the values for each node (key, i) in
52         the lattice
53         # in the form left, right, top, bottom with periodic boundary
54         conditions
55         if i % L == 0:
56             left = i + L - 1

```

```

53         else:
54             left = i - 1
55             if (i + 1) % L == 0:
56                 right = i - L + 1
57             else:
58                 right = i + 1
59             if i - L < 0:
60                 top = i - L + N
61             else:
62                 top = i - L
63             if i + L >= N:
64                 bottom = i + L - N
65             else:
66                 bottom = i + L
67
68             neighbours_dict[i] = [left, right, top, bottom]
69
70     return neighbours_dict
71
72 def get_energy_difference(index_to_flip, s, neighbours_dictionary):
73     """
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
77     neighbours
78     :return: (float) Total energy change of changing that site's spin
79     """
80     sum_of_neighbours = 0
81     for neighbour_index in neighbours_dictionary[index_to_flip]:
82         sum_of_neighbours += s[neighbour_index]
83     total_change = 2*s[index_to_flip]*sum_of_neighbours
84     return total_change
85
86 def metropolis(dE_4, dE_8, N, s, neighbours_dictionary):
87     """
88     The metropolis algorithm as a markov chain monte carlo simulation
89     algorithm that modifies the spin state of the
90     lattice and gives a new state by choosing N (= L x L) sites at random and
91     checking through the energy if it will
92     flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
93     spin will be flipped and the numbers 4, 8
94     represent the corresponding change in energy so that we don't calculate
95     many times an exponential term
96     :param dE_4: (float) Probability for energy change of +4
97     :param dE_8: (float) Probability for energy change of +8
98     :param N: (int) total number of sites
99     :param s: (list) spin list
100     :param neighbours_dictionary: (dict) holds indexes for neighbours
101     :return: (void) does not return anything but changes the state of s
102     """
103     for i in range(N):
104         site_index = randint(0, N-1)
105         dE = get_energy_difference(site_index, s, neighbours_dictionary)
106         rdm_num = uniform(0, 1)
107         if dE <= 0:
108             s[site_index] *= -1
109         elif dE == 4:
110             if dE_4 > rdm_num:
111                 s[site_index] *= -1
112             else:
113                 continue
114         elif dE == 8:
115             if dE_8 > rdm_num:
116                 s[site_index] *= -1
117             else:
118                 continue
119
120 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
171     :param tau: (int) time lag which is the input to the autocovariance
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
176                             range(len(M_list) - tau)]
177     return np.average(autocovariance_list)
178
179 def get_autocorrelation(M_list, tau):
180     """
181     :param M_list: (list) holding average magnetisation for each sweep

```

```

181     neglected pre-thermalised samples
182     :param tau: (int) time lag which is the input to the autocovariance
183     formula
184     :return: (float) autocorrelation for the time lag tau
185     """
186     A_0 = get_autocovariance(M_list, 0)
187     A_tau = get_autocovariance(M_list, tau)
188     return A_tau/A_0
189
190 def get_target_value_index(autocorrelation_list, target_value):
191     """
192     :param autocorrelation_list: (list) autocorrelation for different tau
193     values
194     :param target_value: (float) autocorrelation initial value * 1/e
195     :return: (int) index for target tau
196     """
197     target_index = -1
198     for i in range(len(autocorrelation_list)):
199         if autocorrelation_list[i] < target_value:
200             target_index = i
201             break
202     else:
203         continue
204     return target_index
205
206 def get_target_tau(avg_mag_list):
207     """
208     :param avg_mag_list: (list) holds total magnetisation for each sweep
209     state
210     :return: (int) tau that makes autocorrelation fall to 1/e
211     """
212     autocorrelation_list = []
213     tau_list = np.arange(0, 50, 1)
214
215     for tau in tau_list:
216         autocorrelation_list.append(get_autocorrelation(avg_mag_list, tau))
217
218     target_value = 1/np.e
219     index = get_target_value_index(autocorrelation_list, target_value)
220     target_tau = tau_list[index]
221
222     return target_tau
223
224 def simulation():
225     """
226     Generates data for chi vs T for different values of L along with standard
227     deviation for chi using
228     bootstrap_susceptibility. The data are written in a txt file and plotted
229     using another file called
230     susceptibility_plotter.py.
231     :return: (void)
232     """
233     T_val = np.linspace(1, 4, 51)
234     T_values = [round(T_val[i], 3) for i in range(len(T_val))]
235     L_values = [16, 24]
236     print(f'Total samples to calculate: {len(T_values)*len(L_values)}')
237     n_bins = 100
238     thermalisation_sweeps = 5000
239     sample_every = 80
240
241     with open('susceptibility.txt', 'w') as file:
242         for L in L_values:
243             chi_list, sigma_chi_list = [], []
244             N = L ** 2
245             s = get_spin_list(hot_start, N)
246             neighbours_dictionary = get_neighbours_index(N)

```

```

243         for T in T_values:
244
245             start = time.process_time()
246             mag_list = []
247             b = 1 / T # Constant: 1 / temperature
248             dE_4 = exp(-4 * b) # Probability for energy change of +4
249             dE_8 = exp(-8 * b) # Probability for energy change of +8
250
251             for sweep in range(nsweeps):
252
253                 metropolis(dE_4, dE_8, N, s, neighbours_dictionary)
254
255                 if sweep < thermalisation_sweeps:
256                     continue
257                 else:
258                     if sweep % sample_every == 0:
259                         mag_list.append(get_average_magnetisation(N, s))
260
261                 target_tau = get_target_tau(mag_list)
262                 chi, sigma_chi = bootstrap(mag_list, n_bins, T, target_tau)
263                 chi_list.append(chi)
264                 sigma_chi_list.append(sigma_chi)
265                 time_for_sample = time.process_time() - start
266
267                 file.write(f'{L},{T},{chi},{sigma_chi}\n')
268                 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')
269
270 simulation()
271 susceptibility_plotter()

```

Code file: 3d_lattice.py

```

1 from random import uniform, randint
2 import random
3 from math import exp
4 from lattice_3d_plotter import lattice_3d_plotter
5 import numpy as np
6 import time
7
8 # Global variables
9 H = 0 # Applied magnetic field strength
10 mu = 1 # Magnetic moment
11 M = 0 # Total magnetisation
12 J = 1 # Exchange energy
13 number_of_neighbours = 6 # Number of neighbours in 3D
14 random_seed = 10 # Fix random seed in spin initialization for reproducibility
15 hot_start = False # Initialize with hot start or not
16 nsweeps = 4000 # Number of sweeps
17
18 def get_spin_list(is_hot, N):
19     """
20     :param is_hot: (bool) If is_hot is true make hot start otherwise make
cold start
21     :param N: (int) N = L x L x L, total number of sites
22     :return: (list) containing +/- 1 representing the spins on the lattice
23     """
24     random.seed(random_seed)
25     L = int(N**(1/2))
26     # A hot start means start with +/- 1 with a 50-50 chance on each site
27     if is_hot:
28         s_local = []
29         for i in range(L**3):
30             rdm_num = uniform(0, 1)
31             if rdm_num > 0.5:
32                 s_local.append(1)
33     else:

```

```

34         s_local.append(-1)
35     # A cold start means start with 1s on all sites
36     else:
37         s_local = [1] * (L**3)
38     return s_local
39
40 def get_neighbours_index(N):
41     """
42     :param N: N = L x L
43     :return: (dictionary) key is site value is list of neighbour indices
44     """
45     L = int(N**(1/2))
46     neighbours_dictionary = {}
47     lattice_indices = [[0 for k in range(N)] for j in range(L)]
48
49     for i in range(L):
50         for j in range(N):
51             lattice_indices[i][j] = j + i*N
52     for i in range(L):
53         for j in range(N):
54
55             idx = j + i*N
56
57             if idx % L == 0:
58                 left = lattice_indices[i][j + L - 1]
59             else:
60                 left = lattice_indices[i][j - 1]
61             if (idx + 1) % L == 0:
62                 right = lattice_indices[i][j - L + 1]
63             else:
64                 right = lattice_indices[i][j + 1]
65             if idx - L < 0:
66                 top = lattice_indices[i][j - L + N]
67             else:
68                 top = lattice_indices[i][j - L]
69             if idx + L >= N:
70                 bottom = lattice_indices[i][j + L - N]
71             else:
72                 bottom = lattice_indices[i][j + L]
73             if i == 0:
74                 up = lattice_indices[L-1][j]
75             else:
76                 up = lattice_indices[i-1][j]
77             if i == L-1:
78                 down = lattice_indices[0][j]
79             else:
80                 down = lattice_indices[i+1][j]
81
82             neighbours_dictionary[idx] = [left, right, top, bottom, up, down]
83     return neighbours_dictionary
84
85 def get_energy_difference(index_to_flip, s, neighbours_dictionary):
86     """
87     :param index_to_flip: (int) the site index to consider flipping its spin
88     :param s: (list) spin list of the lattice
89     :param neighbours_dictionary: (dict) holds indexes for each site's
90     neighbours
91     :return: (float) Total energy change of changing that site's spin
92     """
93     sum_of_neighbours = 0
94     for neighbour_index in neighbours_dictionary[index_to_flip]:
95         sum_of_neighbours += s[neighbour_index]
96     total_change = 2*s[index_to_flip]*sum_of_neighbours
97     return total_change
98
99 def metropolis(N, s, neighbours_dictionary, b):
100     """
    :param N: (int) total number of sites

```



```

101 :param s: (list) spin list
102 :param neighbours_dictionary: (dict) holds indexes for neighbours
103 :param b: (float) 1/Temperature
104 :return: (void) does not return anything but changes the state of s
105 """
106 L = int(N**(1/2))
107 for i in range(L**3):
108     site_index = randint(0, (L**3)-1)
109     dE = get_energy_difference(site_index, s, neighbours_dictionary)
110     val = exp(-dE * b)
111     rdm_num = uniform(0, 1)
112     if dE <= 0:
113         s[site_index] *= -1
114     elif val > rdm_num:
115         s[site_index] *= -1
116     else:
117         continue
118
119 def get_magnetisation(N, s):
120     """
121     :param N: (int) total number of sites
122     :param s: (list) spin list
123     :return: (float) Total magnetisation
124     """
125     magnetisation_total = 0
126     L = int(N**(1/2))
127     for i in range(L**3):
128         magnetisation_total += s[i]
129     return magnetisation_total
130
131 def get_average_magnetisation(N, s):
132     """
133     :param N: (int) total number of sites
134     :param s: (list) spin list
135     :return: (float) Magnetisation per site
136     """
137     L = int(N**(1/2))
138     return abs(get_magnetisation(N, s))/(L**3)
139
140 def get_energy(N, s, neighbours_dictionary):
141     """
142     :param N: (int) total number of sites
143     :param s: (list) spin list
144     :param neighbours_dictionary: (dict) holds indexes for neighbours
145     :return: (float) Total energy through the Hamiltonian
146     """
147     sum1 = 0
148     sum2 = 0
149     L = int(N**(1/2))
150     for i in range(L**3):
151         for j in range(number_of_neighbours):
152             sum1 += s[i]*s[neighbours_dictionary[i][j]]
153             if H != 0:
154                 sum2 = get_magnetisation(N, s)
155     total_energy = (-J*sum1 - mu*H*sum2)/2
156     return total_energy
157
158 def get_average_energy(N, s, neighbours_dictionary):
159     L = int(N**(1/2))
160     return get_energy(N, s, neighbours_dictionary)/(3*(L**3))
161
162 def simulation():
163
164     with open('lattice_3d.txt', 'w') as file:
165
166         # Temperature values in Kelvin
167         T_values = np.linspace(2, 6, 51)
168         # Dimension values

```

```

169     L_values = [12, 14]
170     # Sampling
171     thermalisation_sweeps = 2000
172     sample_every = 50
173
174     for L in L_values:
175
176         N = L**2
177         s = get_spin_list(hot_start, N)
178         neighbours_dictionary = get_neighbours_index(N)
179
180         for T in T_values:
181
182             start = time.process_time()
183             b = 1 / T # Constant: 1 / temperature
184             tmp_magnetisation = []
185             tmp_energy = []
186
187             for i in range(nsweps):
188                 metropolis(N, s, neighbours_dictionary, b)
189                 if i < thermalisation_sweeps:
190                     continue
191                 elif i % sample_every == 0:
192                     tmp_magnetisation.append(get_average_magnetisation(N,
193 s))
194                     tmp_energy.append(get_average_energy(N, s,
195 neighbours_dictionary))
196
197                     time_for_sample = time.process_time() - start
198                     file.write(f'{L},{T},{np.average(tmp_magnetisation)},{np.
199 average(tmp_energy)}\n')
200                     print(f'L = {L}, T = {T:.2f}, <m> = {np.average(
201 tmp_magnetisation):.5f}, <E> = {np.average(tmp_energy):.5f}, Time for
202 sample = {time_for_sample:.2f} seconds')
203
204 simulation()
205 lattice_3d_plotter()

```

Code file: deep_ising_temp.py

```

1 from random import uniform, randint
2 import random
3 from math import exp
4 import matplotlib.pyplot as plt
5 import numpy as np
6 from keras.models import Sequential
7 from keras.layers import Dense, Dropout
8 from keras.models import load_model
9 from keras.layers import Conv2D
10 from hyperopt import hp
11 from keras.utils import plot_model
12
13 # Global variables
14 J = 1 # Exchange energy
15 L = 10 # Lattice dimension
16 H = 0 # Applied magnetic field strength
17 mu = 1 # Magnetic moment
18 M = 0 # Total magnetisation
19 number_of_neighbours = 4
20 hot_start = True # Initialize with hot start or not
21 nsweps = 500 # Number of sweeps
22
23 def get_spin_list(is_hot, N):
24     """
25     :param is_hot: (bool) If is_hot is true make hot start otherwise make
26     cold start
27     :param N: (int) N = L x L, total number of sites
28     :return: (list) containing +/- 1 representing the spins on the lattice

```

```

28 """
29 # A hot start means start with +/- 1 with a 50-50 chance on each site
30 if is_hot:
31     s_local = []
32     for i in range(N):
33         rdm_num = uniform(0, 1)
34         if rdm_num > 0.5:
35             s_local.append(1)
36         else:
37             s_local.append(-1)
38 # A cold start means start with 1s on all sites
39 else:
40     s_local = [1] * N
41 return s_local
42
43 def get_neighbours_index(N):
44 """
45 :param N: (int) N = L x L, total number of sites
46 :return: (dict) containing as keys the index of the site on the lattice
47         and as values a list containing the indexes
48         of its neighbours
49 """
50 neighbours_dict = {}
51 L = int(N**(1/2))
52 for i in range(N):
53     # store index of neighbours in the values for each node (key, i) in
54     # the lattice
55     # in the form left, right, top, bottom with periodic boundary
56     # conditions
57     if i % L == 0:
58         left = i + L - 1
59     else:
60         left = i - 1
61     if (i + 1) % L == 0:
62         right = i - L + 1
63     else:
64         right = i + 1
65     if i - L < 0:
66         top = i - L + N
67     else:
68         top = i - L
69     if i + L >= N:
70         bottom = i + L - N
71     else:
72         bottom = i + L
73     neighbours_dict[i] = [left, right, top, bottom]
74
75 return neighbours_dict
76
77 def get_energy_difference(index_to_flip, s, neighbours_dictionary):
78 """
79 :param index_to_flip: (int) the site index to consider flipping its spin
80 :param s: (list) spin list of the lattice
81 :param neighbours_dictionary: (dict) holds indexes for each site's
82         neighbours
83 :return: (float) Total energy change of changing that site's spin
84 """
85 sum_of_neighbours = 0
86 for neighbour_index in neighbours_dictionary[index_to_flip]:
87     sum_of_neighbours += s[neighbour_index]
88 total_change = 2*s[index_to_flip]*sum_of_neighbours # Works out from the
89 Hamiltonian of the before and after states
90 return total_change
91
92 def metropolis(dE_4, dE_8, N, s, neighbours_dictionary):
93 """
94 The metropolis algorithm as a markov chain monte carlo simulation

```

```

91     algorithm that modifies the spin state of the
92     lattice and gives a new state by choosing N (= L x L) sites at random and
93     checking through the energy if it will
94     flip the site's spin or not. The dE_4 and dE_8 are the 2 cases when the
95     spin will be flipped and the numbers 4, 8
96     represent the corresponding change in energy so that we don't calculate
97     many times an exponential term
98     :param dE_4: (float) Probability for energy change of +4
99     :param dE_8: (float) Probability for energy change of +8
100     :param N: (int) total number of sites
101     :param s: (list) spin list
102     :param neighbours_dictionary: (dict) holds indexes for neighbours
103     :return: (void) does not return anything but changes the state of s
104     """
105     for i in range(N):
106         site_index = randint(0,N-1)
107         dE = get_energy_difference(site_index, s, neighbours_dictionary)
108         rdm_num = uniform(0, 1)
109         if dE <= 0:
110             s[site_index] *= -1
111         elif dE == 4:
112             if dE_4 > rdm_num:
113                 s[site_index] *= -1
114             else:
115                 continue
116         elif dE == 8:
117             if dE_8 > rdm_num:
118                 s[site_index] *= -1
119             else:
120                 continue
121
122 def get_magnetisation(N, s):
123     """
124     :param N: (int) total number of sites
125     :param s: (list) spin list
126     :return: (float) Total magnetisation
127     """
128     magnetisation_total = 0
129     for i in range(N):
130         magnetisation_total += s[i]
131     return magnetisation_total
132
133 def get_energy(N, s, neighbours_dictionary):
134     """
135     :param N: (int) total number of sites
136     :param s: (list) spin list
137     :param neighbours_dictionary: (dict) holds indexes for neighbours
138     :return: (float) Total energy through the Hamiltonian
139     """
140     sum1 = 0
141     sum2 = 0
142     for i in range(N):
143         for j in range(number_of_neighbours):
144             sum1 += s[i]*s[neighbours_dictionary[i][j]]
145             if H != 0:
146                 sum2 = get_magnetisation(N, s)
147     total_energy = (-J*sum1 - mu*H*sum2)/2
148     return total_energy
149
150 def get_average_magnetisation(N, s):
151     """
152     :param N: (int) total number of sites
153     :param s: (list) spin list
154     :return: (float) Magnetisation per site
155     """
156     return abs(get_magnetisation(N, s))/N
157
158 def simulation():

```

```

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

```

```

220 def thermalize(T):
221     N = L**2
222     s = get_spin_list(hot_start, N)
223     neighbours_dictionary = get_neighbours_index(N)
224     b = 1 / T # Constant: 1 / temperature
225     dE_4 = exp(-4 * b) # Probability for energy change of +4
226     dE_8 = exp(-8 * b) # Probability for energy change of +8
227     for _ in range(nsweeps):
228         metropolis(dE_4, dE_8, N, s, neighbours_dictionary)
229
230     return np.array([s])
231
232 # predictions = [round(float(model.predict(thermalize(1.9))), 2) for _ in
233     range(5000)]
234 # plt.axvline(x = 1.9, linestyle = '--', color = 'r')
235 # plt.hist(predictions, color = 'k', bins = 59)
236 # plt.title('Histogram for DNN prediction on T = 1.9')
237 # plt.ylabel('Frequency of prediction')
238 # plt.xlabel('Temperature')
239 # plt.savefig('deep_histogram2.pdf', bbox_inches = 'tight')
240 # plt.show()

```