Investigating the behaviours of the Ising Model of a Ferromagnet

Head of Class: Professor David Buscher

Cavendish Laboratory, University of Cambridge
Project produced between: 4/04/22 – 30/04/22
Report written: 30/04/22
Word Count: 2926

Abstract

Investigation of the Ising Model for ferromagnetic spin-spin interactions was achieved computationally through use of the Metropolis algorithm. Efficiencies to performance of the algorithm were managed through use of a vectorised and multithreaded implementation which enabled simulations of larger lattices at much quicker run times. The vectorised approach was a magnitude of 10^2 faster than the traditional approach. Conducting many repeat simulations and evolving these in parallel, it was discovered that when in a random state initially and in the absence of an external field, increasing temperature corresponds to an increase in the likelihood for equilibration of the system. At the critical temperature, the system takes much longer to equilibrate and below this temperature, there is an increased likelihood that the system fails to equilibrate and instead oscillates between metastable states of aligned regions. The value for critical temperature, T_c could be found through maximising the absolute second order derivative of mean magnetisation against temperature. Finding the values for T_c with a linear scaling of lattice size allowed for determination of a true value of $T_c = (2.25 \pm 0.01)J/k_b$ and a value for the critical exponent $\nu = 1.1 \pm 0.1$. Both values are within less than 2 errors from the literature showing good agreement and awarding confidence in the vectorised implementation of the Ising Model. Systems were also scrutinised under differing external magnetic fields which revealed a broadening of heat capacity curves with higher external B-field values and a linear relationship with T_c. Hysteresis effects were additionally observed when altering the B-field at each time step of evolution of the system with overall energy dissipation being inversely dependent on temperature of the system.

1. Introduction

1.1 The Ising Model

The Ising model comprises a mathematical model of lattice sites that can occupy one of two spin states representing magnetic dipole moments[1]. These sites are capable of interacting with their neighbours to flip spins in an attempt to minimise the overall energy of the system. However, heat disturbs the push towards energy minimisation through spin alignment[2]. This model is interesting as it allows for investigation of ferromagnetic behaviours as ferrogmangetism itself arises when a collection of atomic spins align yielding a macroscopic net magnetic moment[3]. The Ising model has been solved analytically in two dimnesions and has many applications in statistical physics[4].

The two dimensional model involves a lattice with $N \times N$ sites that can have a spin value σ of either 1 or -1. Spin flipping is governed by the total energy of the system and hence its Hamiltonian, H which in turn depends on an external field term and an interaction term between neighbouring sites.

$$\boldsymbol{H} = -\sum_{\langle i\ j\rangle} J_{ij}\sigma_i\sigma_j - \mu\sum_j h_j\sigma_j$$

J represents the interaction energy, μ the magnetic moment and h the external field strength. i and j are site positions that must be immediately adjacent to each other. To prevent fiddling around with the $J\ and$ μ parameters, it is possible to measure the total system energy, the magnetic field strength, and the temperature in units of $J, \mu\ and k_b$.

1.2 The Metropolis Algorithm

In a ferromagnetic Ising Model, spins move towards alignment with configurations with aligned adjacent spins

being more probable. The Metropolis algorithm is a Markov chain Monte Carlo Method [5] that is used to determine the time evolution of the spin configurations of the Ising Model. The Metropolis algorithm is analogous to a guided random walk through phase space. We walk along lattice sites randomly and calculate whether the spin of that site should be flipped. The spin is flipped if the total energy to flip the spin $\Delta E < 0$ or if $e^{\frac{-\Delta E}{k_b T}} > p$ with p a random probability [6]. A single timestep evolution of the system occurs when each site is visited in this random walk. Evolving the system many times allows for relevant properties to be determined from the changing spin configuration.

1.3 Vectorisation

The traditional implementation of the Metropolis algorithm conducts this walk site by site which is inefficient and requires $O(N^2)$ operations per timestep. Instead, a modified version of the algorithm can be used which takes advantage of vectorised computation and multithreaded processors [7] to achieve a constant number use of $O(N)$ vector processes. This lower bound for operational complexity allows for computational analysis involving larger lattice sizes and longer time evolutions.

1.4 The Invetstigation

The critical temperature T_c is a point in which Ising Model systems exhibit a large divergence in measured quantities such as heat capacity and mean magnetisation fraction [4]. Analytical solutions of the two-dimensional Ising Model provided by Onsager [4] give a value for T_c . A linear scaling function for T_c against lattice size N can also be assumed. N is the length of one axis of the lattice.

$$T_c(N) = T_c(\infty) + aN^{\frac{-1}{v}}$$

The literature values from Onsager's analysis are given as $T_c(\infty) = 2.27J/k_b$ and $v = 1$.

The behaviour of the Ising Model will be investigated using this modified vectorised Metropolis algorithm. Initially, in the absence of an external magnetic field: equilibration, heat capacity and finite-size scaling will be examined. Dynamics with $h \neq 0$ will also be probed allowing for study of hysteresis effects. Computational estimates of $T_c(\infty)$ and v will be found and compared to the literature values presented in Onsager's model [4] and similar computational analyses [8]

2. Computational Analysis and Implementation

2.1 The Checkerboard Algorithm

Vectorisation is not only exploited in implementation of the Metropolis algorithm through use of vectorised operations, but also in the visiting of lattice sites.

Conventionally, it is necessary to consider flipping the spin of sites one by one. This is because a spin flip of a site causes a change in the evaluation of the Hamiltonian for neighbouring sites. Flipping spins of all sites simultaneously would lead to oscillatory behaviour and no ferromagnetic equilibration of the system. Nevertheless, it is possible to speed up visiting each site through application of the knowledge that only immediate neighbours affect the energy calculation of a site.

Propose a checkerboard mesh was applied on top of a lattice. If a site was under a black section of board, it is assigned an 'odd' parity and if under white, an 'even' parity. From this mesh, it is revealed that sites always have immediate neighbours of opposite parity. As only immediate neighbours affect spin flip calculations, we can confidently flip the spins of all sites with the same parity simultaneously knowing that within the same parity, spin flip calculations at each site are independent of each other. Thus, we can visit sites of each parity simultaneously, reducing the number of flip visits per time step from N^2 to 2.

The above approach is called the 'Checkerboard Algorithm' and produces identical results to the traditional Metropolis Algorithm [9].

The Checkerboard algorithm is implemented in an object-oriented way to allow for effective reuse of the code necessary for the initialisation of an Ising Model system. The CheckerboardIsingModel class was written as a class that stores the necessary fields and methods for system creation.

2.2 Spin flip

The spin state of the system is encoded in a 2D integer numpy array of size $N \times N$ with σ_{ij}being the spin at (i, j). A checkboard mesh for the parities is created as a string numpy array of 'odd' and 'even' elements. As the algorithm requires cyclic boundary conditions, this requires N to be even. An exception is thrown, and an error logged if there is an attempt to pass an odd N to the class constructor.

At each timestep, the odd parity sites are visited first. An alignment matrix, A_{ij} is produced that gives the number of its

neighbours that are spin aligned at a site. Alignment is determined through use of the bitwise XOR function.

σ_{ij}	$\sigma_{i\text{-}1j}$	$\sigma_{ij} \oplus \sigma_{i\text{-}1j}$	$\text{int}(\sigma_{ij} \oplus \sigma_{i\text{-}1j}) + 2$
-1	-1	0	2
-1	1	-2	0
1	-1	-2	0
1	1	0	2

So, the number at site (i,j) corresponds to 2x whether $\sigma_{i+1j}, \sigma_{ij+1}, \sigma_{i\text{-}1j}$, $\sigma_{ij\text{-}1}$ and σ_{ij} are aligned. The energy change for spin flipping ΔE_{ij} is then calculated using the Ising Model Hamiltonian, which for an individual site becomes:

$$\Delta E_{ij} = 2\text{h} - 2\text{h}(\sigma_{ij} + 1) + 2A_{ij} - 8$$

Notice 1 is added to every integer -1,+1 value in σ_{ij} as only positive spins need be considered in the above version of the Hamiltonian. The spins are then flipped according to the relevant ΔE_{ij} criteria discussed in the background and σ_{ij} updated. A check is conducted and if the values of σ_{ij} are not either -1,+1 then an exception is raised indicating something has gone awry in the calculations.

The even parity sites are then swept through. After this, the energy and magnetisation fraction values are calculated and stored in an array with index indicating the timestep that that value was calculated on. Thus, these energy and magnetisation arrays give the evolution of those values over time.

2.3 Multithreading and Parallelisation

This Ising Model is a system with high levels of complexity and is very much dependent on random probabilities in spin flipping that leads to each simulation being individually unique. Thus, it is more useful studying a collection of many systems rather than just the individual one. Constructing CheckerboardIsingModel objects iteratively and time evolving each one sequentially would be inefficient as it would involve a hefty amount of code and program execution would be secluded to use of a single core. It would be faster to make use of multiple cores [7] to parallelise computations and time evolve systems simultaneously across multiple threads.

Parallelisation of computation was achieved through creation of the MultithreadedIsingModel class which takes an array of Ising Models and time evolves them simultaneously through use of the open source multiprocessing python package and its Pool() function. Pool() allows for the creation of worker processes that are handled in parallel by different cores meaning we can assign each core the task of time evolving a system. This increases code execution efficiency.

2.4 Lowess smoothing and curve fitting

Due to the random nature of the evolution of the Ising Model, there are a lot of blips and anomalies in the data that need to be smoothed out. The statsmodels.nonparametric.lowess() [11] function was used and implements a locally weighted scatterplot smoothing that uses local linear interpolation to remove blips in data.

For fitting the function for T_c both scipy.curve_fit was used and numpy.poly_fit. The latter required rewriting the function linearly using logarithms and applying some weighting to prevent bias to small numbers.

2.5 Pickle

As Ising Model computation is expensive, once systems were time evolved, critical values like their magnetisation arrays were stored using pickle serialisation [12]. This allowed for modifications to plots to be made using the same data and without having to constantly rerun the checkerboard algorithm sequence.

2.6 Time speedup and performance

Time for execution was measured using the time package(). It was discovered that use of the combination of multiprocessing and vectorised operations yielded a speedup on the magnitude of $\sim 10^2$. Creation and time evolution of 100 Ising Model systems for 100 time steps, with the same parameters, took 14 secs multithreaded and 16 minutes without! The longest script using the MultithreadedIsingModel class took 40 minutes, showing parallelisation to be a necessity for a performance boost.

However, this performance boost only takes effect if many models are being simulated. The multithreaded approach is ~ 2 slower for fewer than 10 models. Hence, an exception is thrown if there is an attempt to pass only a single model to the MultithreadedIsingModel class and the error logged advises use of a large number of models to enable parallelisation efficiency.

All code was written in the Pycharm IDE [10] and can be fully seen in the appendix.

3. Results and Analysis

3.1 Equilibration

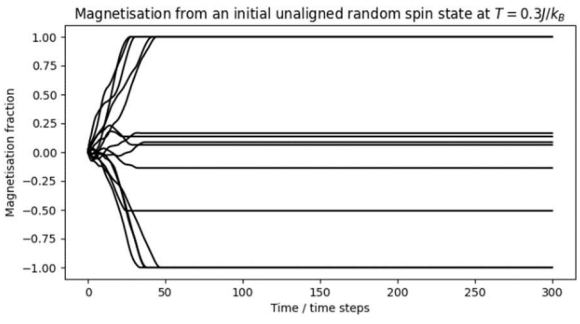
Initially, we examine the dynamics of how a system reaches an equilibration state in the absence of a magnetic field. Equilibration is determined by the magnetisation fraction of the system. Those that reach an absolute fraction value of 1 and hence have a lattice of all aligned spins are in equilibrium.

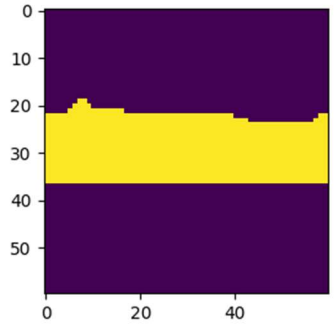
Figure 1a shows that over time, systems tend toward equilibration. However, we notice some systems tend towards a metastable state with formation of magnetic domains of differing spin alignment as seen in 1b. These metastable states comprise the lattice being trapped in local minima. Neither of the opposing spin domains can collapse without significant increase in energy.

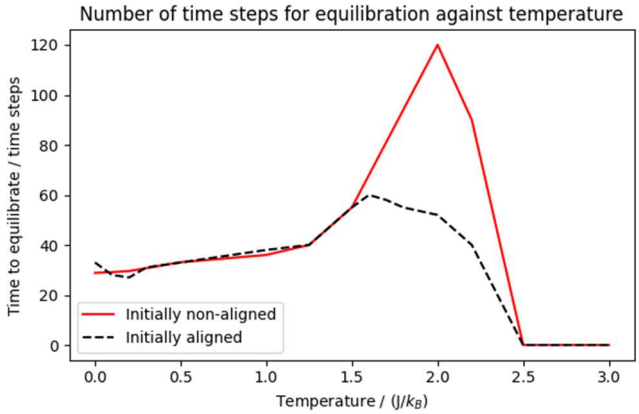
A system was said to be in equilibrium when the standard deviation of magnetisation during a 15 timestep interval was less than twice the RMS fluctuation. This was calculated after allowing sufficient estimated time for equilibration for a similar system at the same temperature. We can see that for future calculations, to allow for equilibration, it is advisable to calculate from the 150th timestep onwards.

From 1c, we see that equilibration time increases steadily until a sudden jump happens converging towards T_c. Above T_c, the system depends less on magnetic domain formation for equilibrium as the spin flip criteria are more easily satisfied. It is seen that in a fully aligned initial state, near the T_c, equilibration time is less as there are fewer domains to collapse.

Figure 1d indicates that there is a lower likelihood of reaching equilibration near T_c and hence a higher likelihood of being trapped in a metastable state. Higher temperature above T_c have more ability through thermal fluctuations to collapse magnetic domain bands into alignment.





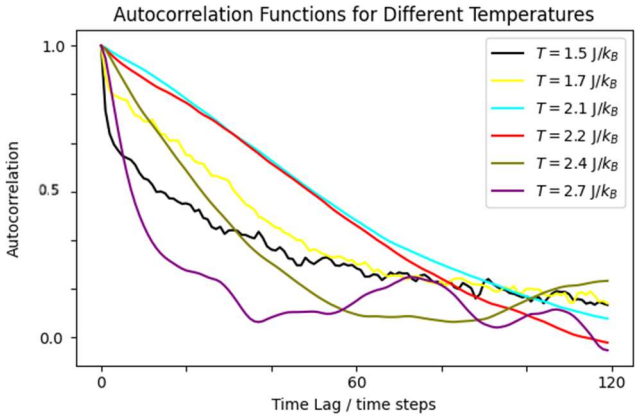


Temperature / J/k B	0	0.5	1	1.5	2	2.4	3
% Equilibrated	65	67	75	80	95	75	100

Figure 1. Equilibration properties against temperature for a system of size N=60. (a) Magnetisation over time. (b) Snapshot of a metastable state that failed to reach equilibration. (c) Equilibration time. (d) Table showing fraction of systems that reach equilibration.

3.2 Autocorrelation

The autocorrelation function of magnetisation was then studied for 1000 timesteps and evaluating the function after the 150th timestep to allow for equilibration. Figure 2a shows that the autocorrelation function at different temperatures follow an exponential decay shape which is as expected from theory. Decorrelation time was then calculated in Figure 2b using appropriate lag times. It is seen that the decorrelation times peak near the value of T_c. Larger lattice sizes have a narrower decorrelation peak and are shifted towards larger temperatures. More temperature increments and measurements were taken around the peak at T_c to ensure accurate curve representation due to the rapid change.



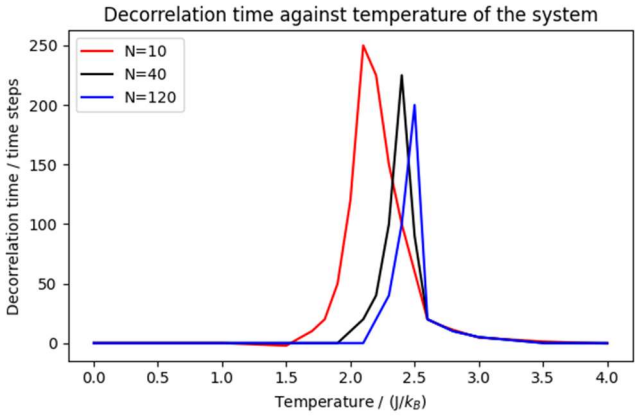


Figure 2. (a) Autocorrelation against time lag for a system of size N=40. (b) Decorrelation time against temperature for several lattice sizes.

3.3 Temperature Dependence and Critical Temperature

We now review the temperature dependence relationships of the Ising model with a particular focus on the heat capacity and dynamics near T_c.

Using 500 temperature increments for 4 different lattice sizes and evolving these over 1000 timesteps. It's seen in Figure 3a that magnetisation fraction rapidly drops to 0 at T_c. The rapid drop off can be modelled as a polynomial of the form $M = M_0 T^a$. Truncating the data within the bounds of the drop off and rewriting the model linearly as

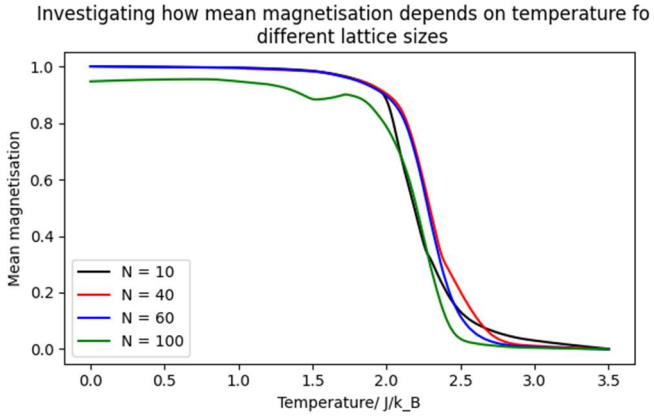
$\ln m = \ln M_0 + a \ln T$ allows for use of numpy.poly_fit to find the parameters and plot the fit. The fit shown in Figure 3b closely resembles that of Onsager's theoretical result [4].

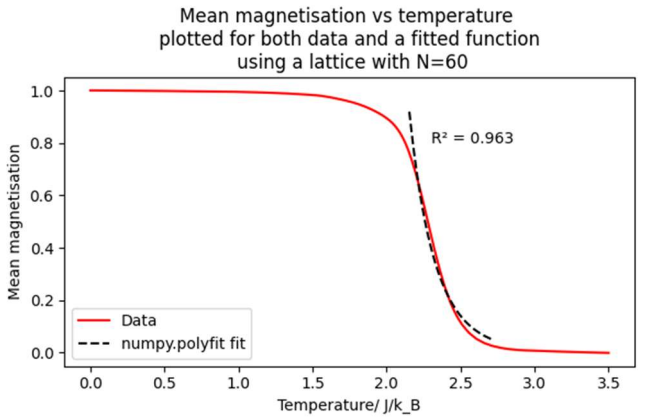
The derivative of the magnetisation is also calculated along with a fit in Figure 3c. Maximising the absolute value of the derivative of magnetisation gives the value for T_c which by definition occurs at the point of most rapid change of magnetisation.

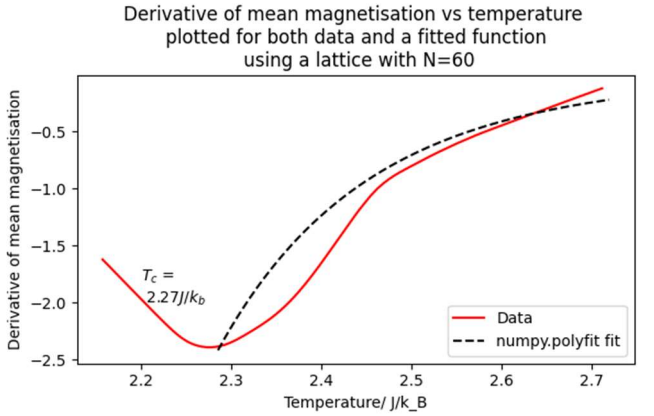
Figure 3d shows energies increasing with increasing temperature but showing a rapid increase at T_c. Heat capacity was calculated both computationally through numerical differentiation and through use of the fluctuation dissipation theorem.

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

The close agreement between the two curves in Figure 3e verifies that the theorem holds for this system.







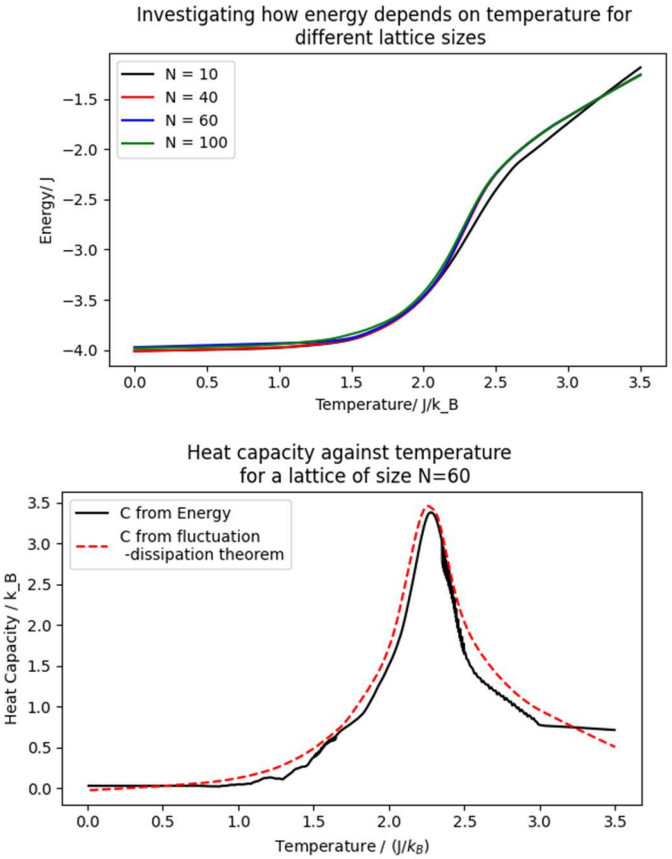


Figure 3. Dynamics of systems in equilibrium against temperature for systems with no external magnetic field. (a) Magnetisation fraction for different lattice sizes. (b) Magnetisation fraction fit. (c) Derivative magnetisation fraction and critical temperature. (d) Energy for different lattice sizes. (e) Heat capacity through numerical differentiation and through fluctuation-dissipation theorem.

3.4 Finite-size Scaling

We then explore linear finite-size scaling on the value of T_c with an increasing value of N of the form:

$$T_c(N) = T_c(\infty) + aN^{\frac{-1}{\nu}}$$

Critical temperatures were calculated for several N values using the maximisation of the absolute mean magnetisation against temperature derivative. A scipy curve_fit fit was then found to allow for the calculation of the critical variables in the scaling relationship. Figure 4b shows the fit and the data and the value of $T_c(\infty)$ the data converges upon.

Fitted Variable	Value	Error in Value	% Difference From theory	Error Difference From Theory
$T_c(\infty)$	2.25	0.01	0.85	1.92
a	1.9	0.4	-	-
v	1.1	0.1	10	1

An exponent of $\nu = 1.1 \pm 0.1$ was fitted and compared to a theoretical value of $\nu = 1$. A $T_c(\infty)$ value of $T_c(\infty) = 2.25 \pm 0.01\, J/k_b$ was fitted and compared to the theoretical Onsager result of $T_c(\infty) = 2.27\, J/k_b$. Both values are within two errors of theory. Hence, the computationally acquired values show good agreement with Onsager's analysis [4] and indeed with other similar computational analysis [8], demonstrating an alignment between computational methods and theory. The main sources in error were the somewhat arbitrary cut-off values used in isolating the magnetisation fraction drop off near T_c that is an error propagated through to fit determination and hence T_c value calculation.

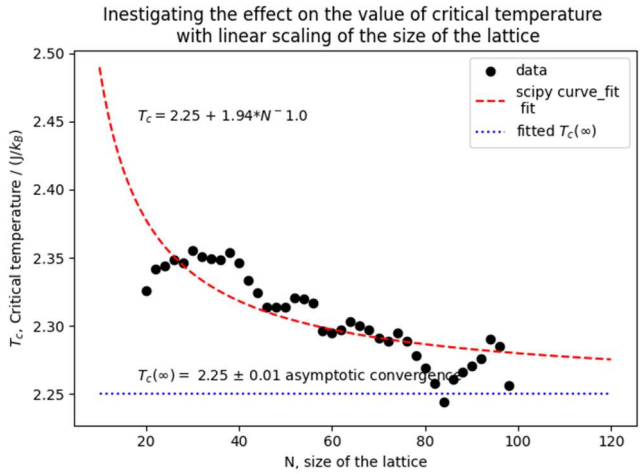


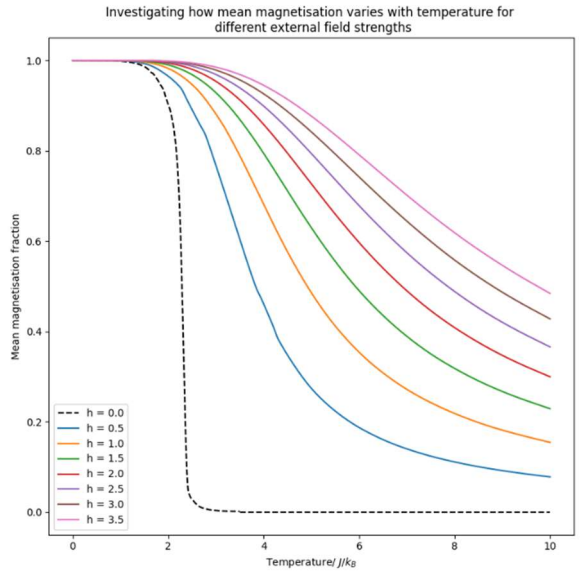
Figure 4. Finite size scaling. (a) Values of the fit determined (b) Critical temperatures against lattice size showing the scaling relationship and the convergence at the infinity value of critical temperature.

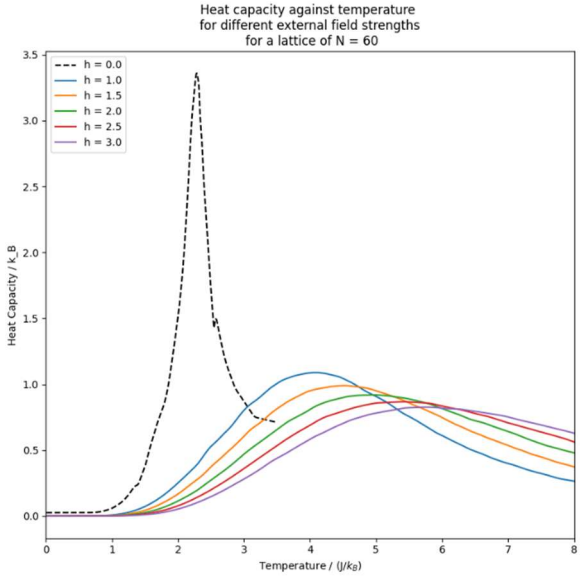
3.5 External field dynamics

Finally, we survey the dynamics of the Ising Model in an external magnetic field. Plotting both magnetisation and heat capacity against temperature in Figure 5 shows that lower values of h have a narrower phase transition and that increasing h shifts these transitions to higher temperatures.

T_c were then calculated for increasing external field strength showing a linear straight-line relationship in Figure 5c. As h increases, T_c increases. The fit was found to be:

$T_c = 1.09h + 2/\ln(1+\sqrt{2})$.





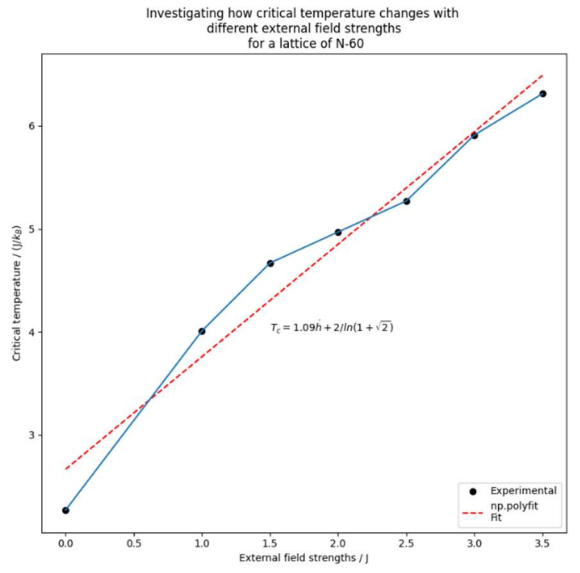
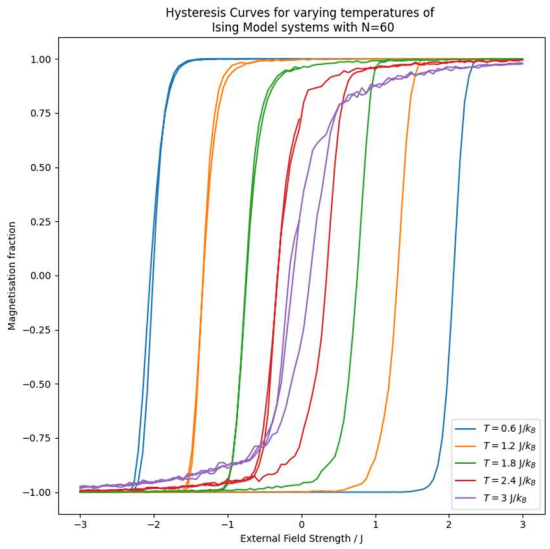


Figure 5. Dynamics in an external magnetic field. Magnetisation, heat capacity and critical temperature of a system in equilibrium with size N=60 against several external magnetic field strengths. (a) Magnetisation (b) Heat Capacity (c) Critical temperature showing straight line relationship with h value

3.6 Hysteresis Effects

Hysteresis effects were also investigated by cycling through the h value. The systems were time evolved one time step at a time. Before each time step, the h value of the system was changed. Steady state magnetisation against h is plotted in Figure 6a showing hysteresis. We see hysteresis starting from a temperature value of $T = 0.6J/k_b$. Below this value, there is not enough energy to perturb the system from it's equilibration state and thus no hysteresis is observed. Past this threshold, the area enclosed in the hysteresis loop and hence its heat dissipation during the cycle goes down. This is because at higher temperatures it is easier and more efficient to perturb these magnetic domains. Heat dissipation spikes at the threshold and decreases with increasing temperature in Figure 6b.



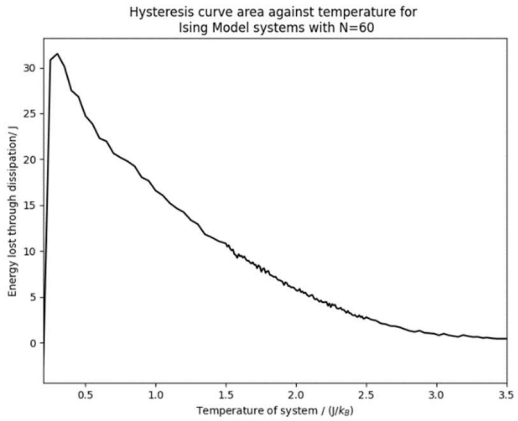


Figure 6. Hysteresis effects. (a) Hysteresis curves from cycling h value at different temperatures (b) Energy dissipation calculated through energy enclosed in the hysteresis loop.

4. Conclusions

4.1 Overview

An investigation was conducted into the behaviours of the 2D Ising Model. Using a vectorised and multithreaded take on the traditional Metropolis algorithm allowed for a $\sim 10^2$ speedup performance boost. The relationship between equilibration, magnetisation and heat capacity against temperature were studied and peaks were witness around the critical temperature. The effects of finite-size scaling on T_c were considered and critical values of $v = 1.1 \pm 0.1$ and $T_c(\infty) = 2.25 \pm 0.01\, J/k_b$ were found, both being within two errors of Onsager's analysis [4] and thus showing good agreement with existing literature. The autocorrelation, effects in an external magnetic field and hysteresis behaviour all were in alignment with what would be expected from scientific theory. Reproduction of theory suggests that the computational method was successful.

4.2 Improvements

Nevertheless, improvements can be made to the method.

The Metropolis algorithm assumes that the walk through lattice sites is ergodic [5]. However, this is only valid if the walk is of infinite duration which is not physically possible. Thus, it is possible that some regions are never visited by the algorithm. This problem can be addressed by frequently restarting the simulation from a random configuration i.e. doing more than one repeat for each plot and averaging the plot repeats [6].

When we time evolved, we always swept through odd than even parities. We failed to consider whether different results occurred for the visiting the parities $even -> odd$ or randomly. We must establish that there is no difference in results caused by this.

The investigation also excludes odd sized lattices due to cyclic boundary conditions; we must establish that these too would yield different results.

5. References

[1] Stanford.edu. 2022. *The Ising Model*. [online] Available at: <https://stanford.edu/~jeffjar/statmech/intro4.html> [Accessed 2 May 2022].

[2] Baierlein, R. (1999), Thermal Physics, Cambridge: Cambridge University Press, ISBN 978-0-521-59082-2

[3] Farside.ph.utexas.edu. 2022. *The Ising model*. [online] Available at: <https://farside.ph.utexas.edu/teaching/329/lectures/node110.html> [Accessed 2 May 2022].

[4] L. Onsager, "Crystal statistics. i. a two-dimensional model with an order-disorder transition," Phys. Rev., vol. 65, pp. 117–149, Feb 1944.

[5] Hastings, W.K. (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". Biometrika. **57** (1): 97-109. Bibcode:1970Bimka..57...97H. doi:10.1093/biomet/57.1.97. JSTOR 2334940. Zbl 0219.65008.

[6] T. Fricke, "Monte Carlo Investigation of the Ising Model" 2006, "Numerical Solutions to the Ising Model Using the Metropolis Algorithm" [online] Available at: <https://www.asc.ohio-state.edu/braaten.1//statphys/Ising_MatLab.pdf> [Accessed 2 May 2022].

[7] Block, Benjamin, Peter Virnau, and Tobias Preis. "Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model." *Computer Physics Communications* 181.9 (2010): 1549-1556.

[8] D. Bennett, "Numerical Solutions to the Ising Model Using the Metropolis Algorithm" [online] Available at: < https://www.maths.tcd.ie/~dbennett/js/ising.pdf> [Accessed 2 May 2022].

[9] K. Yang, Y.-F. Chen, G. Roumpos, C. Colby, and J. Anderson, "High Performance Monte Carlo Simulation of Ising Model on TPU Clusters," 2019

[10] JetBrains, 2017. *PyCharm*. [online] JetBrains. Available at: <https://www.jetbrains.com/pycharm/> [Accessed 11 April 2017].

[11] Seabold S, Perktold J. statsmodels: Econometric and statistical modeling with python. In: 9th Python in Science Conference. 2010.

[12] Van Rossum G. The Python Library Reference, release 3.8.2. Python Software Foundation; 2020.

[13] Blundell, Stephen J.; Blundell, Katherine M. (2009). Concepts in thermal physics. OUP Oxford.

5. Appendix: Main Numerical Code

This appendix provides an overview of the structure of the code.

The main classes and methods are in the files CheckerboardIsingModel.py and MultithreadedIsingModel.py. The constructor for ChechekerboardIsingModel takes in the necessary parametrs and creates an object that is an Ising Model system. The routines in CheckerboardIsingModel such as progress_time_step() and compute_properties() allows for the time evolution of the system whilst determining the necessary properites that are to be investigated. MultithreadedIsingModel creates an object that allows for the parallel time evolutoin of many CheckerboardIsingModel

object simultaneously. More detail about the operation of these classes and methods can be found in the comments and docstrings.

The following table details which scripts prodcued which graphs that are included in which figures in the report.

Script	Graphs generated	Figure
1_equilibration	EQU 1.png EQU 2.png EQU 3.png	Figure 1
2_auto_and_de_correlation	AUTO 1.png AUTO 2.png	Figure 2
3_temperature_dependence	TEMPER 1.png TEMPER 2.png TEMPER 3.png TEMPER 4.png TEMPER 5.png	Figure 3
4_scaling	SCAL 1.png	Figure 4
5_external_field	EXT 1.png EXT 2.png EXT 3.png	Figure 5
6_hysteresis	HYS 1.png HYS 2.png	Figure 6

Implemementation of the Core Numerical Code:

File - C:\Users\leond\PycharmProjects\pythonProject\CheckerboardIsingModel.py

```
# Blind Grade Number: 6905T

import numpy as np
import sys


class CheckerboardIsingModel:
    '''
    Implementation of the Ising Model with associated calculations of states and properties using a checkerboard
    variant of  the Markov Chain Monte Carlo method known as the Metropolis Algorithm. The algorithm has been
    simplified using the benefits of vectorised computation.

    Note that the arguments passed to the constructor are in a form of reduced units and have factors of the exchange
    energy, the magnetic moment and the Boltzmann constant taken out.

    Args passed to the constructor:
        N (int): the total number of lattice points along one axis, thus the system has N^2 total lattices
        h_tilde (float): the strength of the external magnetic field
        T_tilde (float): the temperature of the system

    Attributes:
        N (int): the total number of lattice points along one axis, thus the system has N^2 total lattices
        h_tilde (float): the strength of the external magnetic field T_tilde (float): the temperature of the system
        energy (float): the energy of the system evaluated at the current time step in the reduced units defined above
        energy_array (float[]): array of energies calculated at each time step
        magnetisation (float): the magnetisation of the system evaluated at the current time step in the reduced units
            defined above
        magnetisation_array (float[]): array of the magnetisations calculated at each time step
        spin_configuration (np.int[]): matrix of the spins of each lattice point
        time_step_counter (int): an integer counter containing the number of time steps that the
            system has undergone evolution for
        checkerboard_pattern (string[]) : matrix of the parities of the lattice points, serving as a mesh
        store_configuration (bool): if set to true, stores each configuration in an array
        configuration_array ([np.int[]]): store of the spin configurations at each time step


    Methods:
        progress_time_step(): progresses the system forward by a single time step
        compute_properties(): calculates the magnetisation and energy of the system at that time step
        determine_neighbours(): returns a tuple of the neighbours of a lattice element ordered
            according to the order of the cardinal directions
    '''

    # Creating the constructor for the class
    def __init__(self, N: int, h_tilde: int, T_tilde: float, aligned: bool = True, store_configuration: bool = False):
        '''
        Args: N (int): the total number of lattice points along one axis, thus the system has N^2 total lattices
        h_tilde (float): the strength of the external magnetic field
        T_tilde (float): the temperature of the system
        aligned (bool): boolean indicating whether the initial spin configuration should be all spins aligned (True) or
```

```
51                      randomly aligned (False). Default is aligned.
52          store_configuration (bool): boolean flag indicating whether to store each spin
    configuration at each
53                                      time step; this flag exists as storing each
    configuration would be very
54                                      computationally intensive and require a large amount
     of RAM, thus for this
55                                      reason, it is automatically set to False.
56
57          '''
58          # Creating the fields for the properties / attributes of the object
59          self.h_tilde = h_tilde
60          self.T_tilde = T_tilde
61          self.N = N
62          self.spin_configuration = np.ones((N, N), dtype=int)
63          self.energy = None
64          self.energy_array = []
65          self.magnetisation = None
66          self.magnetisation_array = []
67          self.time_step_counter = 0
68          self.checkerboard_pattern = None
69          self.store_configuration = store_configuration
70          self.configuration_array = []
71
72          # The code is written to only handle lattices with even dimensions for lattice
    number on either two-dimensional
73          # axis. To account for this, an Exception is thrown in the case N is odd. This
    is also logged. N is then changed
74          # to the evn number 1 higher.
75          if N % 2 != 0:
76              self.N += 1
77              print(f"Odd N input. Exception thrown.", file=sys.stderr)
78              raise Exception("This implementation of the Metropolis algorithm only
    handles even lattice dimensions. ")
79
80          # Start on a random alignment if aligned is set to false
81          if not aligned:
82              self.spin_configuration = np.random.choice([-1, 1], (self.N, self.N)).astype
    (int)
83
84          # Create the checkerboard pattern to speed up the Metropolis Algorithm by
    flipping same parity simultaneously
85          self.checkerboard_pattern = np.tile([["odd", "even"], ["even", "odd"]], (int(
    self.N / 2), int(self.N / 2)))
86
87          # Calculate properties at the current time step of the simulation
88          self.compute_properties()
89
90      def determine_neighbours(self):
91          '''
92          Function to determine the neighbours of a lattice point
93          :return: A tuple containing two elements. aligned_spins = the number of aligned
    spins for all lattices totaled.
94                   And aligned_spin_matrix = matrix where value at each lattice point is -
    2x the number of adjacent aligned
95                   spins.
96          '''
97
98          # The bitwise operation for determining whether two lattice points are aligned
    is the XOR gate
99          # which can be implemented using the numpy bitwise functions. The lattice points
     share the same
100         # spin if the resultant matrix position has a 0 element and a different spin
    with a -2 element.
101         # np.roll allows us to get to the correct direction of neighbour using matrix
    shift parameters.
102
```

File - C:\Users\leond\PycharmProjects\pythonProject\CheckerboardIsingModel.py

```
103         north_neighbour = np.bitwise_xor(self.spin_configuration, np.roll(self.
    spin_configuration, 1, 0))
104         east_neighbour = np.bitwise_xor(self.spin_configuration, np.roll(self.
    spin_configuration, -1, 1))
105         south_neighbour = np.bitwise_xor(self.spin_configuration, np.roll(self.
    spin_configuration, -1, 0))
106         west_neighbour = np.bitwise_xor(self.spin_configuration, np.roll(self.
    spin_configuration, 1, 1))
107
108         # Need a matrix that enumerates the number of adjacent aligned spins for each
    individual lattice element
109         # Have to first modify the format of neighbour matrices such that non-aligned
    are 0. Instead of checking each
110         # element against a criteria and changing it, we can do vectorised arithmetic
    which is quicker.
111         # Hence, an aligned spin is now represented by a +2 element.
112         north_neighbour = north_neighbour + 2
113         east_neighbour = east_neighbour + 2
114         south_neighbour = south_neighbour + 2
115         west_neighbour = west_neighbour + 2
116
117         # Find the number of points that have aligned spins. +2 indicates spin alignment
118         aligned_spins = np.count_nonzero(north_neighbour) + np.count_nonzero(
    east_neighbour) \
119                         + np.count_nonzero(south_neighbour) + np.count_nonzero(
    west_neighbour)
120
121         aligned_spin_matrix = north_neighbour + east_neighbour + south_neighbour +
    west_neighbour
122
123         return aligned_spins, aligned_spin_matrix
124
125     def compute_properties(self):
126         """
127         Computes the energy and magnetisation of the lattice.
128         :return: Nothing. Modifies the values of energy and magnetisation of the object
    and appends these values
129                  to the corresponding arrays containing those values.
130         """
131         # Magnetisation calculation and appending to the array of magnetisation at each
    time step
132         self.magnetisation = (2 * np.count_nonzero((self.spin_configuration + 1)) - self
    .N ** 2) / self.N ** 2
133         self.magnetisation_array.append(self.magnetisation)
134
135         # Energy calculation using the formula of the Ising Model
136
137         # Fetch neighbours
138         aligned_spins = self.determine_neighbours()[0]
139         anti_aligned_spins = 4 * (self.N ** 2) - aligned_spins
140
141         positive_spin = np.count_nonzero(self.spin_configuration == 1)
142         negative_spin = self.N ** 2 - positive_spin
143
144         exchange_value = anti_aligned_spins - aligned_spins
145         magnetic_value = self.h_tilde * (positive_spin - negative_spin)
146
147         self.energy = (exchange_value + magnetic_value) / self.N ** 2
148
149         # Energy calculation and appending to the array of energies at each time step
150         self.energy_array.append(self.energy)
151
152     def progress_time_step(self, time_steps: int):
153         '''
154         Progresses the system through a provided number of time steps. A time step is
    taken when the spins are
155         flipped for both parities.
```

File - C:\Users\leond\PycharmProjects\pythonProject\CheckerboardIsingModel.py

```
156
157         :param time_steps: int, number of time steps to progress the system through
158
159         :return: Nothing. The CheckerboardIsingModel object evolved through the
    specified number of time steps with
160                  the fields of the object updated accordingly.
161         '''
162         # Do the necessary evolution actions for the specified number of time steps
163         for _ in range(time_steps):
164
165             # If the store_configuration marker is set to True, store the configuration
    as appropriate
166             if self.store_configuration:
167                 self.configuration_array.append(self.spin_configuration.copy())
168
169             # Creating a matrix of probabilities to use in determining whether a random
    flip in spin occurs
170             probabilities = np.random.random((self.N, self.N))
171
172             # One time step involves two processes, considering both even and odd
    parities.
173             parities = ["odd", "even"]
174
175             # Iterate across the two parities
176             for parity in parities:
177                 aligned_spin_matrix = self.determine_neighbours()[1]
178
179                 # Note here the factor in front of the aligned spins is 2 instead of 4,
    this is because the
180                 # aligned_spin_matrix represents an aligned neighbour as a value of 2
181                 # !!!!!!!!!!!!!!!!!
182                 energy_change = 2 * self.h_tilde - (self.spin_configuration+1) * 2 *
    self.h_tilde + 2 * aligned_spin_matrix - 8
183
184                 # Determining whether the spin of the lattice point should be flipped
185                 if self.T_tilde > 0:
186                     flip_spin_matrix = (np.exp(-energy_change / self.T_tilde) >
    probabilities).astype(np.bool)
187                 else:
188                     flip_spin_matrix = (energy_change < 0).astype(np.bool)
189
190                 # Removing instances where a flip is required for lattice elements of
    the parity not currently being
191                 # considered
192                 if parity == 'odd':
193                     flip_spin_matrix = np.bitwise_and(flip_spin_matrix, self.
    checkerboard_pattern == 'odd')
194                 else:
195                     flip_spin_matrix = np.bitwise_and(flip_spin_matrix, self.
    checkerboard_pattern == 'even')
196
197                 # Flipping the spins accordingly
198                 flip_spin_matrix = flip_spin_matrix.astype(np.int) * -1
199                 flip_spin_matrix[flip_spin_matrix == 0] = 1
200                 self.spin_configuration = flip_spin_matrix * self.spin_configuration
201
202                 # Troubleshooting the contents of the spin configuration field, should
    only be populated with 1 and -1
203                 if np.count_nonzero(self.spin_configuration + 2) != self.N ** 2:
204                     print(f"Spin configuration is populated by something other than 1,-
    1s", file=sys.stderr)
205                     raise Exception("Error in the spin configuration field of the system
    .")
206
207             # Update the time, energy and magnetisation fields of the object
208             self.time_step_counter += 1
209             self.compute_properties()
```

File - C:\Users\leond\PycharmProjects\pythonProject\MultithreadedIsingModel.py

```python
# Blind Grade Number: 6905T

import multiprocessing
import sys
import numpy as np
from CheckerboardIsingModel import CheckerboardIsingModel

class MultithreadedIsingModel:
    '''
    A class that enables for multiple Ising Model systems to be evolved through and considered simultaneously
    across several processor cores in parallel. This is necessary as computing systems sequentially for large
    time steps is computationally intensive. Running these computations with one model per thread allows for
    significant speed up at higher time step values.

     Args passed to the constructor:
        model_array ([CheckerboardIsingModel]): an array of the Ising Model systems that are to be evolved
                                            using multithreading and hence run in parallel by different processor threads

    Attributes:
        model_array ([CheckerboardIsingModel]): an array of the Ising Model systems that are to be evolved
                                            using multithreading and hence run in parallel by different processor threads
        number_of_models (int): the total number of models being evolved in parallel through multithreading,
                                    equal to the length of model_array
    '''

    def __init__(self, model_array: [CheckerboardIsingModel]):
        '''
        The constructor that allows for parallel processing of the time evolution of the models.
        :param model_array: the models that are to be time-evolved in parallel
        '''

        # It is only efficient to use the multithreading class if multiple models are to be
        # investigated. Thus, an exception is thrown and an error is logged in the
        # instance that only a singular model is passed to the constructor

        single_system_flag = False
        if not (isinstance(model_array, list) or isinstance(model_array, np.ndarray)):
            single_system_flag = True

        if not single_system_flag:
            self.number_of_models = len(model_array)
            if self.number_of_models < 2:
                single_system_flag = True

        if single_system_flag:
            # Log an error
            print(f"Using multithreading class with only one system. This is inefficient! "
                  f"Only use MultithreadedIsingModel with multiple CheckerboardIsingModel objects!!!"
                  f"Exception thrown.", file=sys.stderr)

            # Throw an exception
            raise Exception("Do not use MultithreadingIsingModel with one model only.")

        # Set and calculate the necessary class properties accordingly
        self.model_array = model_array
```

File - C:\Users\leond\PycharmProjects\pythonProject\MultithreadedIsingModel.py

```
56          self.number_of_models = len(model_array)
57
58      # Function that does the parallel time evolution of the models
59      def simultaneous_time_steps(self, time_steps: int):
60          '''
61          Runs the multithreading_time_step in parallel with the models stored in
    model_array by
62          using the multiprocessing package and the Pool() function.
63          :param time_steps: the number of time steps the different models should be
    evolved by
64          :return: updates model_array to contain the updated and evolved models
65          '''
66
67          # Use the multiprocessing package to evolve models in parallel
68          # Pool represents a pool of worker processes that are handled simultaneously by
    different processors
69          with multiprocessing.Pool() as pool:
70              # Create a new array of the outputs of passing the models in model_array in
    parallel to the
71              # multithreading_time_step function
72              updated_models = pool.starmap(multithreading_time_step, [(model, time_steps
    ) for model in self.model_array])
73              updated_models = list(updated_models)
74
75              # Update the field model_array with the new evolved models
76              self.model_array = []
77              for i in range(self.number_of_models):
78                  self.model_array.append(updated_models[i])
79
80
81
82  def multithreading_time_step(model: CheckerboardIsingModel, time_steps: int):
83      '''
84      A helper function that takes a CheckerboardIsingModel object and a specified number
    of time steps and
85      evolves the lattice contained in the object by the time steps. This is defined non-
    locally as this is
86      required for the starmap function. Multiple models are passed through this function
    concurrently to
87      allow for multithreading of model evolution. This is the function run simultaneously
     across multiple
88      cores.
89      :param model: CheckerboardIsingModel,the model to be evolved through by the time
    steps
90      :param time_steps: int, the integer number of time steps
91      :return: The updated model object progressed through by the time steps specified.
92      '''
93      model.progress_time_step(time_steps)
94      return model
95
```