Investigating 2D Ferromagnetic Ising Model*

Lingfa Meng†
Cavendish Laboratory, 19 J J Thomson Avenue, Cambridge CB3 0HE, UK.
(Dated: April 27, 2020)

We implemented a python-based Checkerboard algorithm [1] to simulate a 2D Ferromagnetic Ising system. Equilibrium time for an initial state with a binary spin distribution was measured by phase-space clustering of the magnetization time series. Correlation time, equilibrium magnetization were measured for different temperatures. Curie temperature was calculated from specific heat capacity measurements. Error measurements were carried out using the Jackknife algorithm. The simulations were verified against theoretical predictions.

Keywords: Ising Model, Ferromagnet, Checkerboard Algorithm, Jackknife Algorithm, Phase-space Clustering

I. INTRODUCTION

2D Ising Model is the lowest-dimension Ising Model exhibiting an ordered-disordered phase transition with discrete symmetry [2], first solved exactly by Lars Onsanger [3].

Ising models are a simple yet effective model in lattice-based condensed matter systems, such as ferro-magnets and polymer random-walks. It is also used for modeling influenza spreading [4]. This paper aims to verify the validity of this Ising model implementation, and develop better understanding of it through studying hysteresis effects.

Section II provides the theoretical predictions of Ising model. Section III is the general computational methods. Sections IV, V, VI, VII and VIII are separately the methods and results for the equilibrium time, auto-correlation time, equilibrium magnetization, finite size scaling and ferromagnetic hysteresis. Section IX is the discussion of results. Section X is the conclusion.

II. THEORETICAL ANALYSIS

$$E = -J \sum_{<ij>} S_i S_j + \mu H \sum_i S_i \tag{1}$$

2D Ising model with periodic boundary condition (pbc) on a square lattice with a square domain with the Hamiltonian 1 is studied. Interaction strength and magnetic moment data for bcc Fe as given in table 2.1 and 2.2 from [6] are used but the bcc lattice is not used.

A. Curie Temperature

Lars Onsanger [3] predicted Curie temperature for 2D isotropic square lattice to be the following, in units of J/k_B, where J is the spin-spin coupling strength and k_B is the Boltzman constant.

$$T_C = \frac{2}{\ln\left(1+\sqrt{2}\right)} = 2.269185... \tag{2}$$

B. Finite-size Scaling

The finite-size scaling is the system size dependence of phase transition point for the order parameter.

$$T_C(N) = T_C(\infty) + aN^{-1/\nu} \tag{3}$$

$T_C(\infty)$, a and ν are system dependent constants. This relation 3 cannot be linearized in N. We have to fit a non-linear function to recover $T_C(\infty)$.

C. Specific Heat Capacity

The specific heat capacity is derived using the fluctuation-dissipation theorem.

$$C = \frac{{\sigma_E}^2}{k_B T^2} \tag{4}$$

D. Ferromagnetic Hysteresis

Ferromagnetic hysteresis is due to the system being able to remember its evolving path. Strength of the hysteresis for a material is characterized by the switching energy needed per site.

III. COMPUTATIONAL METHOD

A. Monte Carlo: from Metropolis to Checkerboard

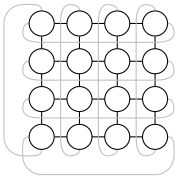

FIG. 1: Periodic boundary condition example.

We perform many Monte Carlo (MC) steps to evolve an isotropic square system with pbc (sides connected to opposite sides forming a torus) 1 into a equilibrium state, from which we can measure physical quantities. A single MC step is updating 1 exactly n^2 sites where n is the lattice size in one dimension.

Algorithm 1 Single site update

Require:

All sites' spins initialized;

Choose a spin to flip;

Ensure:

Calculate ΔE required to flip the spin;

If $\Delta E < 0$, flip the spin;

Else if $\exp(-\Delta E/k_B T) > p$ where p is drawn from a even distribution from $[0, 1]$, flip the spin.

Same updating protocol 1 for single site is adopted in the Metropolis and Checkerboard algorithms.

1. *Metropolis Algorithm*

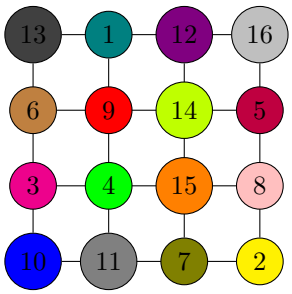

FIG. 2: A randomly updated 2D lattice example, numbering indicates updating order.

The Metropolis algorithm [7] updates n^2 randomly selected sites 2, resulting in n^2 function calls with a costly random number function.

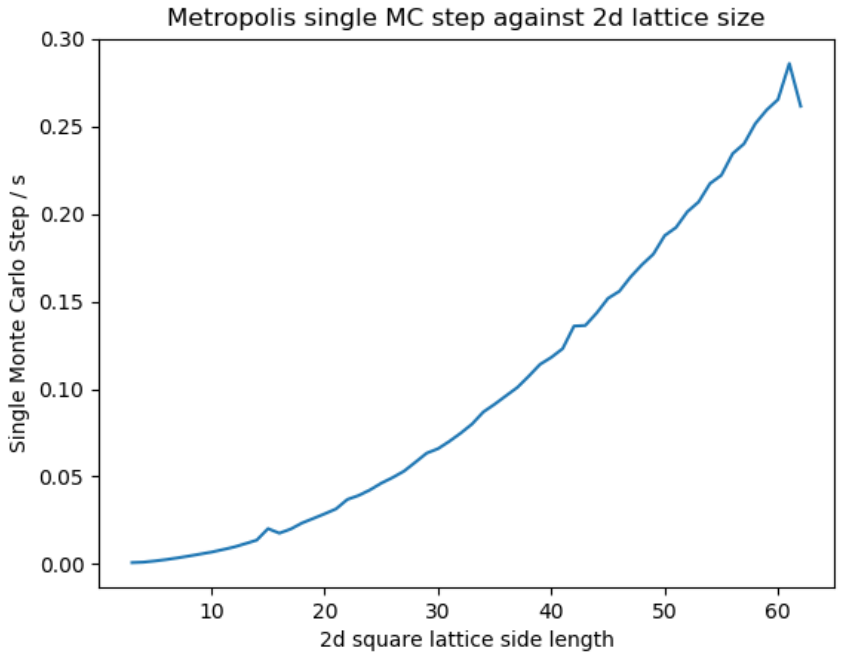

FIG. 3: Performance of Metropolis algorithm.

Time complexity for a single MC step with respect to lattice size n in d dimension is $O(n^d)$ 3.

2. *Checkerboard Algorithm*

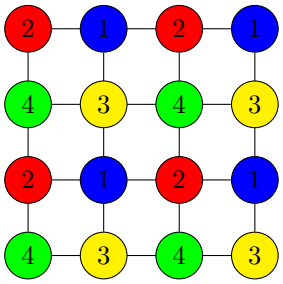

FIG. 4: A 2D Checkerboard example divided into 4 interleaved sub-lattices, sites labeled with a same number are updated simultaneously.

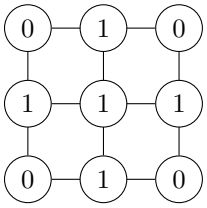

FIG. 5: A binary convolution kernel for calculating local interaction energies.

Metropolis algorithm is improved by choosing an efficient updating procedure. By dividing into 4 interleaved sub-lattices 4 then updating a sub-lattice as a whole, we avoid randomizing and reduce the number of function calls. The Checkerboard algorithm was implemented for arbitrary dimension odd sized Ising system. This paper works with odd sized systems without loss of generality. Local interaction energies are computed by convolving spins locally with a binary kernel 5 representing local interactions.

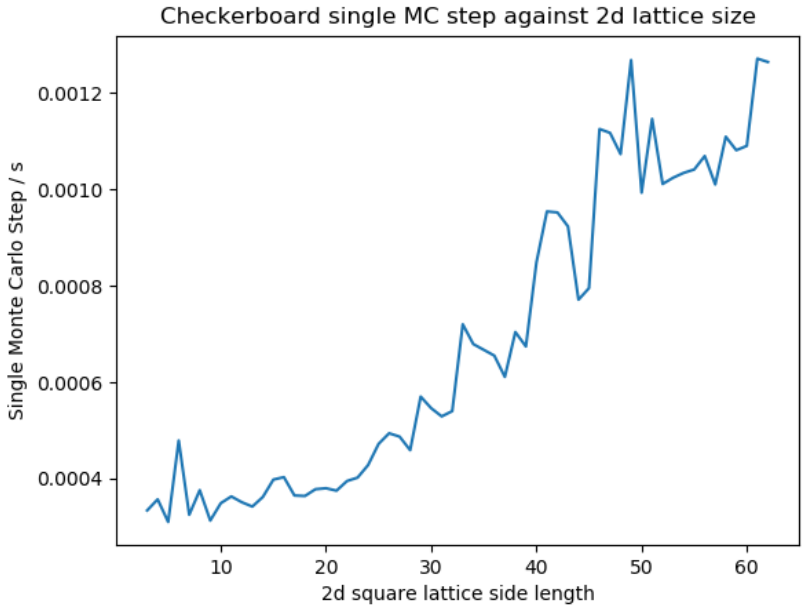

FIG. 6: Performance of Checkerboard algorithm.

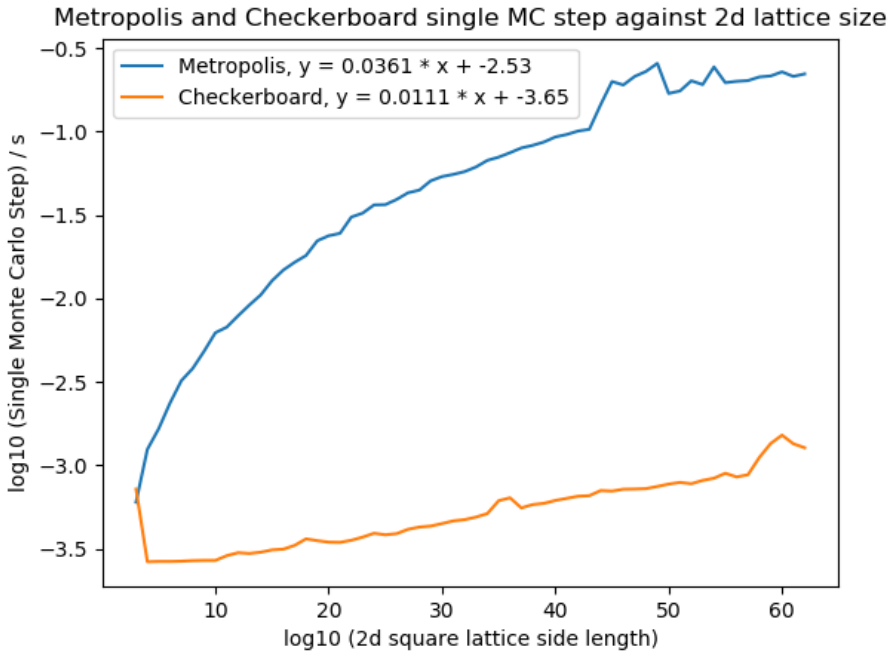

FIG. 7: Performance comparison (log10) between Metropolis and Checkerboard algorithms.

Time complexity for a single MC step with lattice size n and m sites being updated simultaneously by numpy vectorized functions is $O(n^d/m)$ 6. A speedup of more than 100 times is achieved for systems sized between 10 and 60 7, which is the working regime for other sections. This improved computational efficiency enables calculation of larger systems with more accurate error bars in the following sections, especially in simulating hysteresis effect VIII where quantization errors are undesirable.

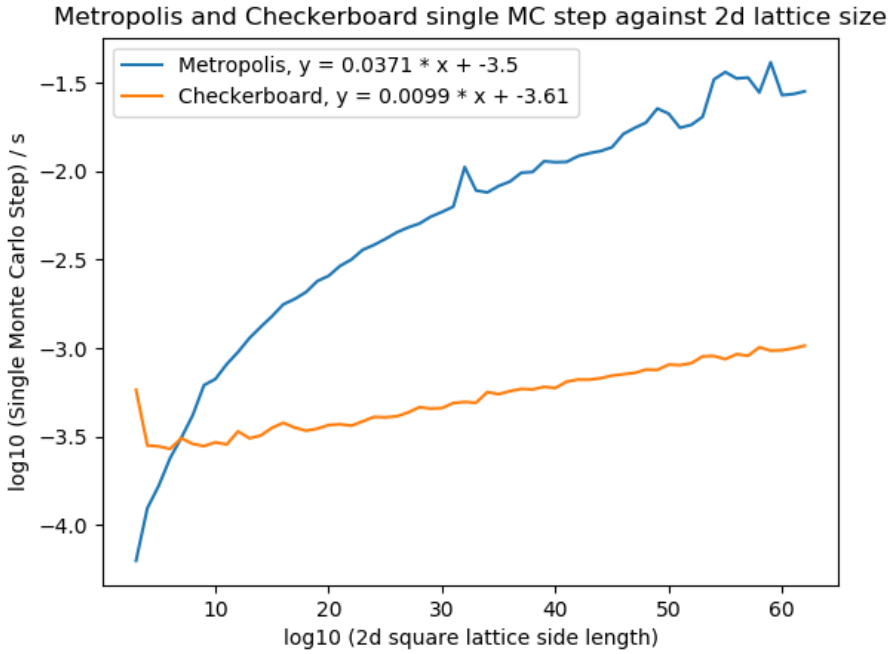

FIG. 8: Performance comparison (log10) between Metropolis (non-random) and Checkerboard algorithms.

The Checkerboard algorithm is still more than 100 times faster for systems sized between 30 and 60 8 compared with non-random Metropolis.

B. Error Estimation

Errors are deduced from repeated measurements. Ising model produces correlated data, therefore Jackknife algorithm is used as opposed to Bootstrap algorithm for non-trivial statistics, e.g. Binder's Cumulent. Standard deviation is used when the desired statistics is mean.

IV. EQUILIBRIUM TIME

A Gaussian clustering algorithm in total magnetization phase-space determines time taken to reach equilibrium state from a random initial state, where each site is equally likely to be spin up or down. Assuming system moves to an equilibrium state eventually, any long enough simulation will have the minority cluster size corresponding to the equilibrium time. This algorithm is applicable to all MC processes where the phase space is localized in equilibrium state.

We will test the algorithm validity and plot equilibrium time against temperature on a 41 by 41 system.

A. Methods

Algorithm 2 Equilibrium state detection

Require:

Long enough (5 times longer than the correlation time) simulated total magnetization time series;

Ensure:

Cluster total magnetization phase space into 2 Gaussian clusters;

If cluster sizes differ by more than half of the simulation time, the minority cluster size is returned as the time taken to reach equilibrium;

Else either system started in or has not reached equilibrium and a longer simulation time may be needed, and returns 0 as equilibrium time by default.

Notice that a default equilibrium time 0 is returned when the algorithm fails to detect any equilibrium state.

B. Results

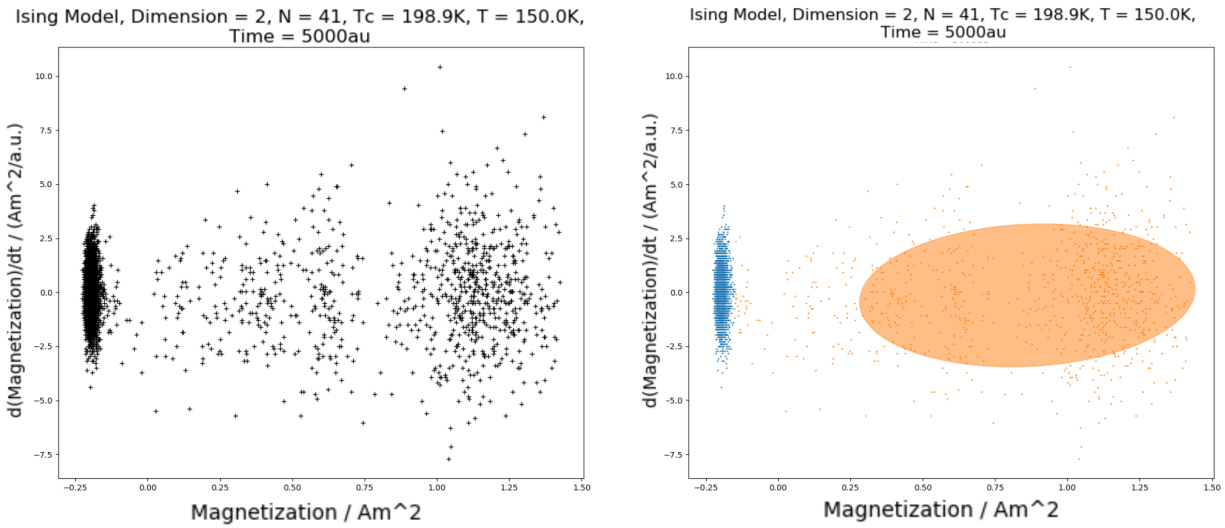

FIG. 9: Magnetization phase space clustering.

Clustering on a system below Curie temperature 9 shows a densely populated equilibrium state cluster in blue, with minority cluster in orange representing other intermediate states in reaching equilibrium.

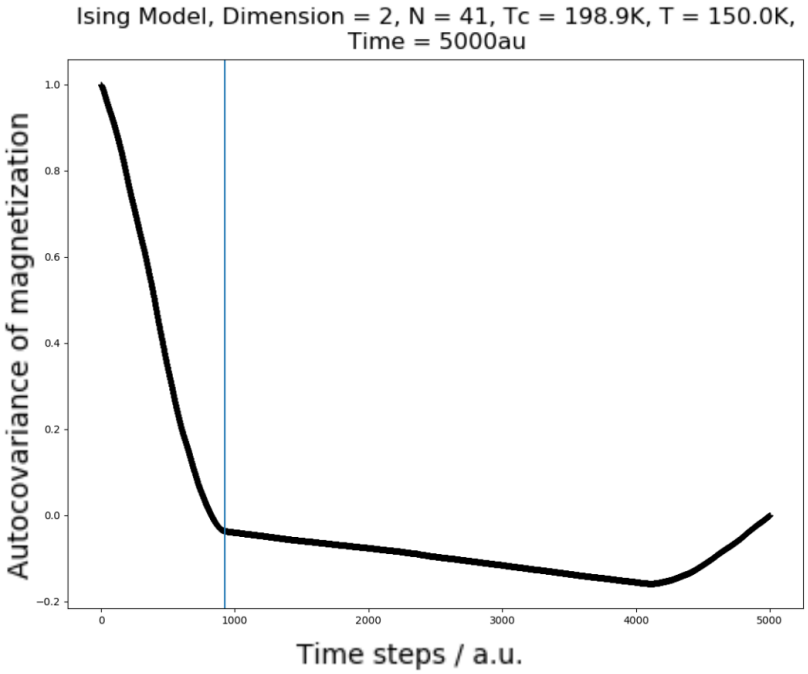

FIG. 10: Equilibrium time (blue line) on auto-covariance against time graph.

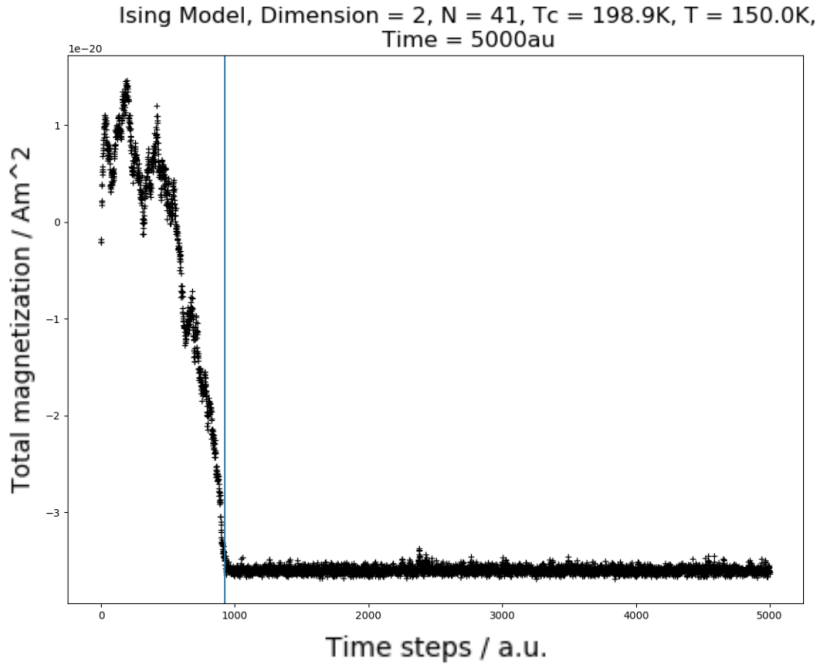

FIG. 11: Equilibrium time (blue line) on magnetization against time graph.

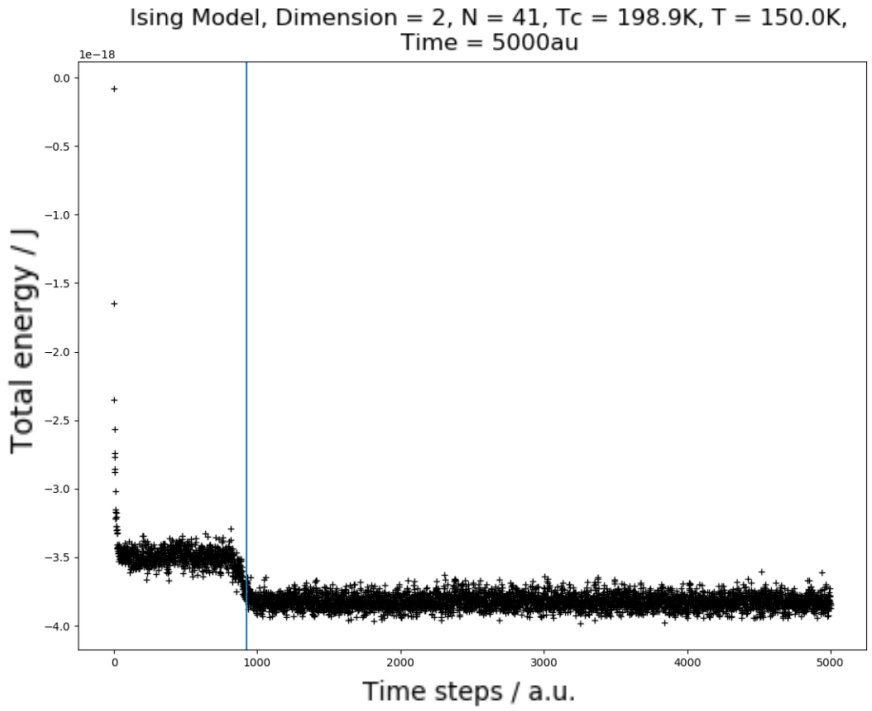

FIG. 12: Equilibrium time (blue line) on total energy against time graph.

From 10, 11 and 12 the estimation of equilibrium time passes the sanity check by noticing the overlapping of the blue line (estimation) and the start of the equilibrium state.

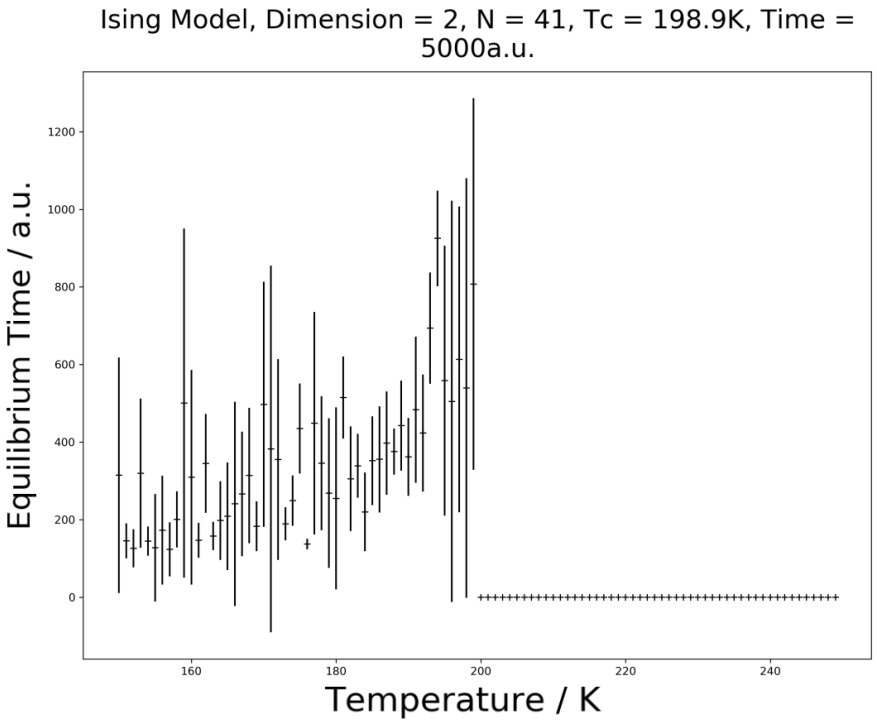

FIG. 13: Equilibrium time against temperature.

Time taken to reach equilibrium depends on the difference between initial state and the equilibrium state, and the correlation time. In the low temperature regime with a short correlation time, the initial state is a hot state (high energy) and far from the equilibrium state. Near and below the Curie temperature with a long correlation length, the system is in a cold state (low energy) and close to the equilibrium state. We observe a growing equilibrium time in temperature below the Curie temperature, implying equilibrium time grows with and is dominated by correlation time.

Above the Curie temperature, system is already in the equilibrium state and equilibrium time is zero.

V. CORRELATION TIME

Correlation time is the amount of time the system needs to evolve such that the initial state becomes irrelevant in the case of pure thermal fluctuations. When averaging to measure any physical quantities, we must average over more than at least 2 correlation time to get the result to reduce random error. We will study the correlation time for systems of sizes 11, 15 and 19, focusing on near Curie temperature behaviour.

A. Methods

Algorithm 3 Auto-correlation time

Require:

All sites' spins initialized;

Ensure:

Evolve the system for more than 5 times of the correlation time;

Determine and discard non-equilibrium evolution section of the time series;

Compute the auto-correlation for the remaining magnetization time series, normalized by auto-correlation at $\Delta t = 0$;

Identify time taken for normalized auto-correlation to drop below $1/e$ as the correlation time;

Single measurement was carried out using 3, and repeated measurements were taken to measure the error.

B. Results

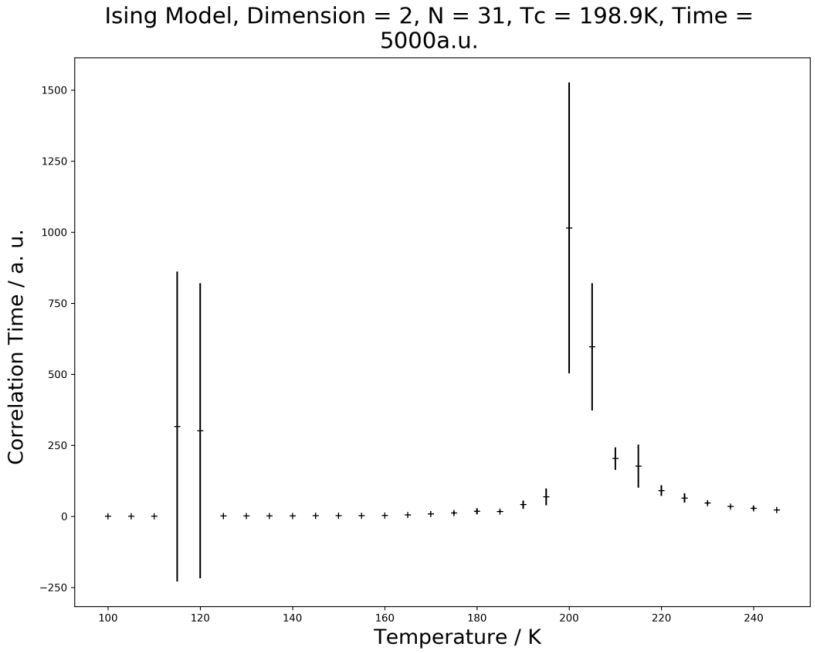

FIG. 14: Correlation time against temperature over ordered and disordered phases.

The singular point in correlation time is around $200K$. We zoom in onto that section for systems of size 11, 15 and 19.

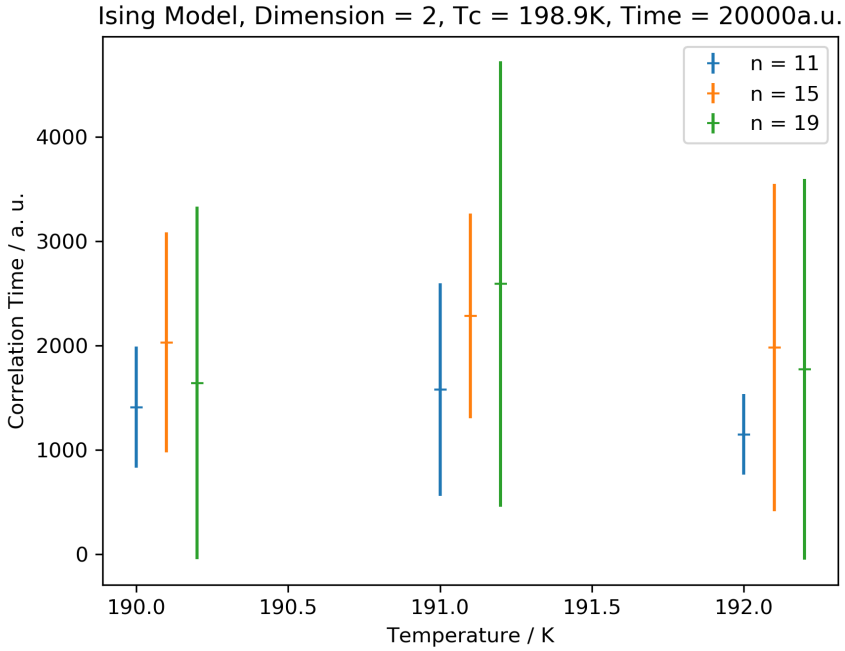

FIG. 15: Correlation time for systems of different sizes around the Curie point.

Correlation time increases from around 1000 to 2000 as the system size increases 15. Curie point is located around the highest correlation time, at $191K$. This implies that at least 10000 MC steps is needed for systems sized between 11 and 21 near the Curie temperature. It is allowed to run for less MC steps for off Curie point simulations.

Calculating correlation time is longer than measuring most physical quantities, we effectively choose an upper bound (e.g. 10000 MC steps) and does not calculate the correlation time every time when making measurements.

VI. EQUILIBRIUM MAGNETIZATION

We will study the temperature dependence of the equilibrium magnetization for systems of size 31, focusing on near Curie temperature behaviour.

A. Method

Algorithm 4 Equilibrium magnetization

Require:

All sites' spins initialized;

Ensure:

Evolve the system for more than 10 times of the correlation time;

Determine and discard non-equilibrium evolution time series;

Return mean of the remaining magnetization time series.

Single equilibrium magnetization measurement is carried out using 4. Repeated measurements gives an estimate for the error.

B. Result

Scanning the whole temperature domain helps find the phase transition point of the order parameter.

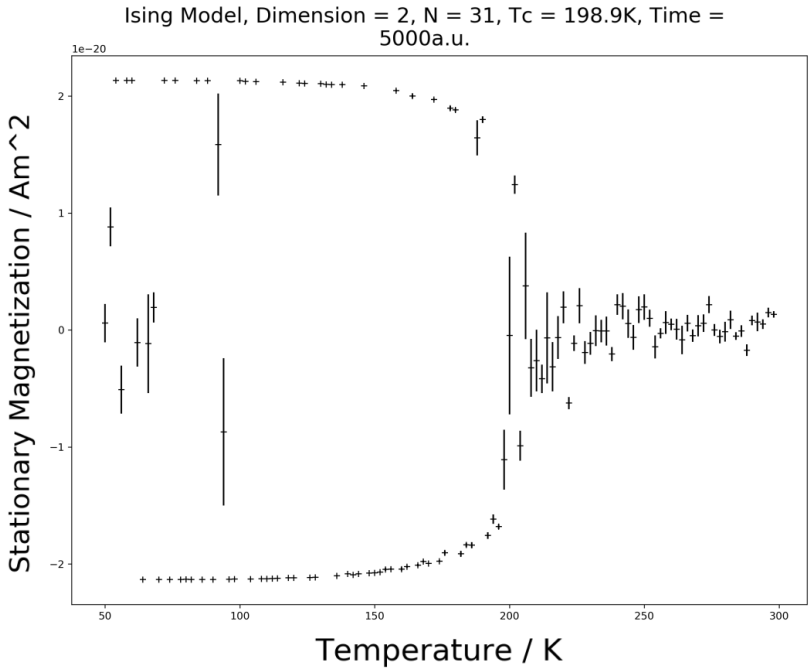

FIG. 16: Equilibrium magnetization for ordered and disordered phases.

High temperature regime exhibits disordered phase while low temperature regime exhibits ordered phase 16. 8 outliers near zero magnetization at low temperature are due to system being unable to move out of local energy minima by insufficient thermal fluctuations. We notice that thermal fluctuations is strongest around the theoretical Curie temperature $T_C = 198.9K$ 2.

A phase transition is observed above 200K. We zoom in on 200K to 250K region to study the phase transition behaviour.

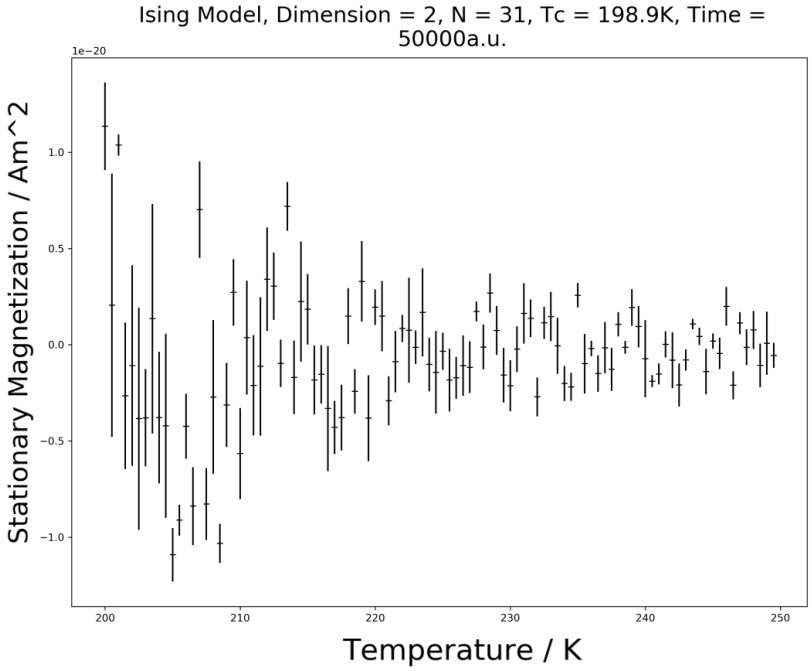

FIG. 17: Equilibrium magnetization with Curie point details.

A transition from ordered phase into disordered phase is observed around 228K. Anomalously low thermal fluctuation was observed for the equilibrium magnetization around the phase transition point. This is possibly due to the finite size effect displacing the phase transition point of the order parameter from the singularity point of the correlation time.

VII. FINITE SIZE SCALING OF CURIE TEMPERATURE FROM SPECIFIC HEAT CAPACITY

We measure the singular point for specific heat capacity, and non-linearly fit the data using scipy.optimize.curve_fit according to 3 to obtain $T(\infty)$ for the finite size scaling.

A. Method

Algorithm 5 Specific Heat Capacity

Require:

Initialize all spins;

Ensure:

Evolve the system for more than 5 times of the correlation time;

Compute $C = \frac{{\sigma_E}^2}{k_B T^2}$ 4 as the specific heat.

Algorithm 6 Determine Curie Temperature for given system size

Require:

Determine the range to look for Curie temperature;

Determine an array of points to explore within that range, spaced ΔT.

Run the system for 10 times as long as the longest correlation time present among the points in the array.

Ensure:

Compute Specific heat capacity for all points in the array;

Point with the largest specific heat capacity is the predicted Curie point, where the error is over-estimated as ΔT.

B. Result

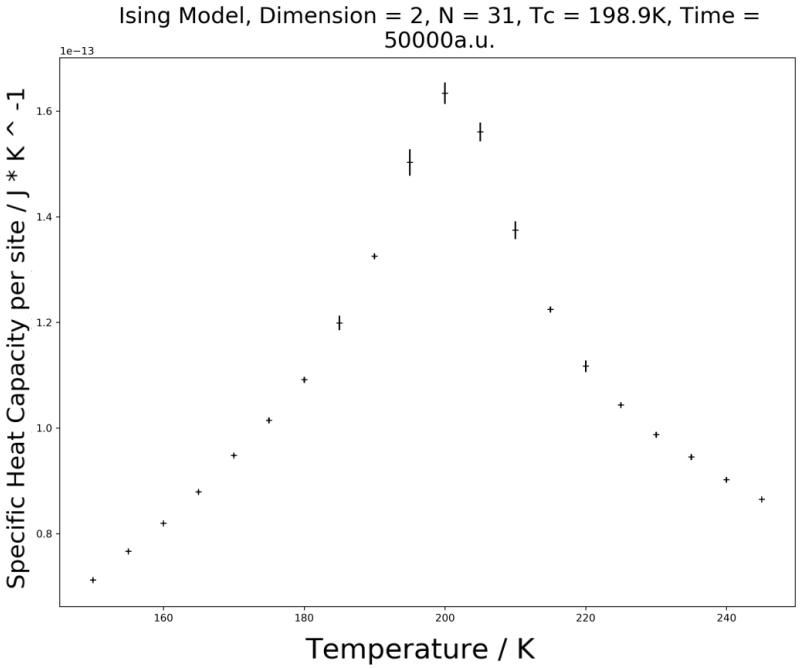

FIG. 18: Specific heat capacity for 31 by 31 system.

By scanning the relevant temperature range, the Curie point is located around $200K$. This reduces the searching domain size for T_C. We proceed to calculate the Curie point within $190K$ to $210K$.

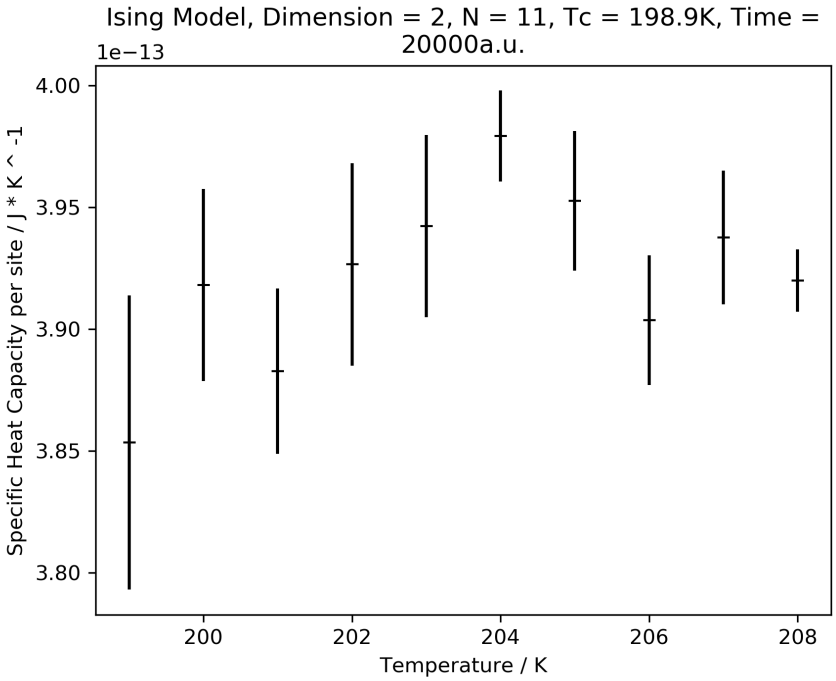

FIG. 19: Specific heat capacity for 11 by 11 system near Curie point.

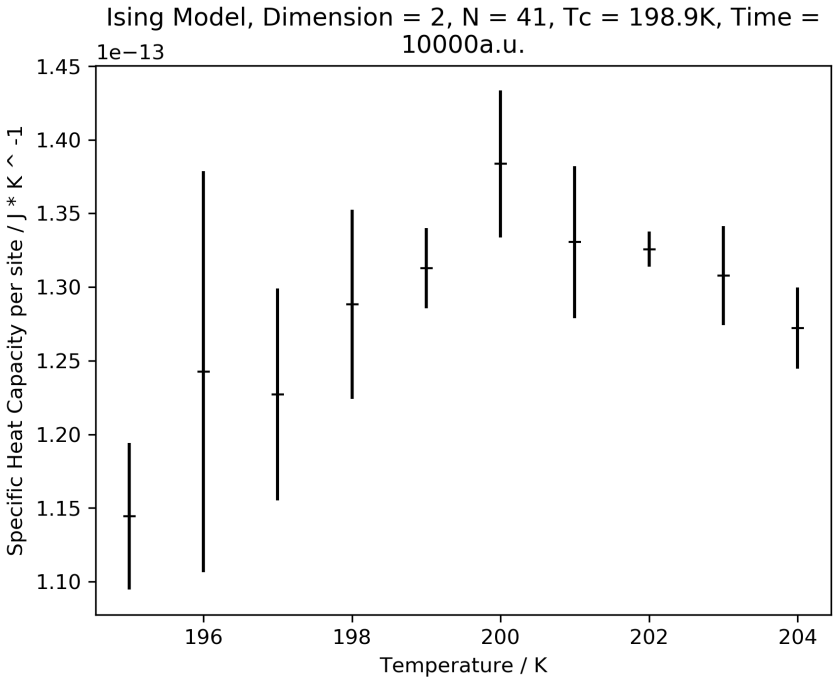

FIG. 20: Specific heat capacity for 41 by 41 system near Curie point.

Curie temperatures for systems between size 41 and 11 lie between $199K$ 20 and $206K$ 19. We proceed to plot Curie temperatures against sizes between 11 and 41.

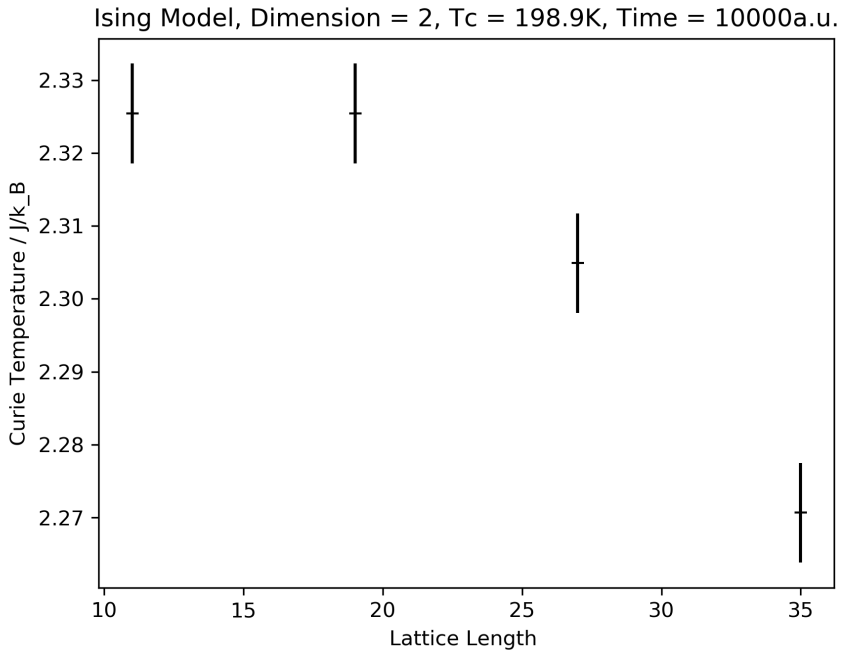

FIG. 21: Curie temperature against system size.

Our upper estimate (according to the scaling relation 3) is $T_C = 2.270 \pm 0.007$ 21, with theoretical prediction $T_C = 2.269$ 2 lying just within 1 standard deviations. The data verified that Curie temperature decreases as systems size increases according to the finite-size scaling relation 3.

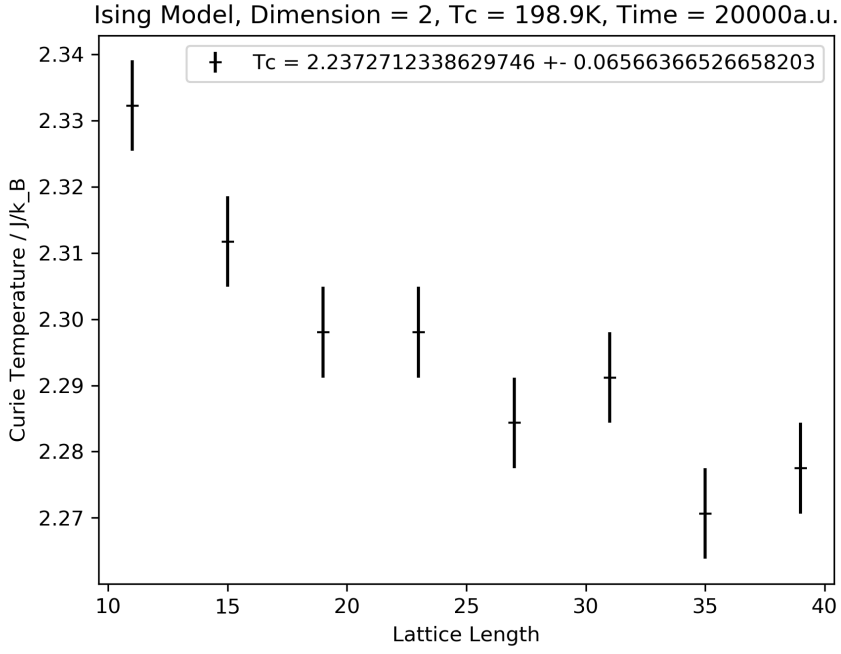

FIG. 22: Finite size scaling of Curie temperature against system size.

By fitting 3 using least square procedure (scipy.optimize.fit_curve) and considering the error in Curie tempera-

tures, we estimated $T_C(\infty) = 2.24 \pm 0.07$ 22. Theoretical prediction lies within 1 standard deviation, justifying the correctness of the simulation.

A down shift in simulated Curie temperature is observed for both $T_C(specific\,heat) = 196 \pm 6$ and $T_C(correlation\,time) = 191\,\mathrm{K}$ as compared to theoretical prediction $T_C(theory) = 198.9\,\mathrm{K}$.

VIII. HYSTERESIS SIMULATION

Hysteresis effect appears when the system undergoes cyclic load (external field). The Dissipation energy per site for switching the Ising model is plotted against temperature.

A. Method

Algorithm 7 Ferromagnetic Hysteresis

Require:

All sites' spins initialized from a binary distribution;

Ensure:

Evolve the system for a time $1.25 * T$, where T is the period of a sawtooth wave time varying external magnetic field;

Extra 0.25 cycle is simulated to remove initial state. Prolonged hysteresis loops can be produced using 7 with more periods. The external field's time dependence in a cycle is given below.

$$\mathrm{H(t)}= \begin{cases} -H_0 \times t, & \text{for } 0 \leqslant t \leqslant T/4 \\ -H_0 \times (T/2 - t), & \text{for } T/4 \leqslant t \leqslant T/2 \\ +H_0 \times (t - T/2), & \text{for } T/2 \leqslant t \leqslant 3T/4 \\ +H_0 \times (T - t), & \text{for } 3T/4 \leqslant t \leqslant T \end{cases}$$

Algorithm 8 Measuring Dissipation Energy

Require:

System evolved for 0.25 + n cycles of length T;

Ensure:

Excluding the initial 0.25 cycle;

Set dissipation energy E = 0;

Within each remaining cycle, Action(t)= $\begin{cases} E+ = -M(t) \times H_0/(T/4), & \text{for } 0 \leqslant t \leqslant T/4 \\ E+ = -M(t) \times H_0/(T/4), & \text{for } T/4 \leqslant t \leqslant T/2 \\ E+ = +M(t) \times H_0/(T/4), & \text{for } T/2 \leqslant t \leqslant 3T/4 \\ E+ = +M(t) \times H_0/(T/4), & \text{for } 3T/4 \leqslant t \leqslant T \end{cases}$

Return $\frac{E}{n*N^2}$ where N is the lattice size.

B. Result

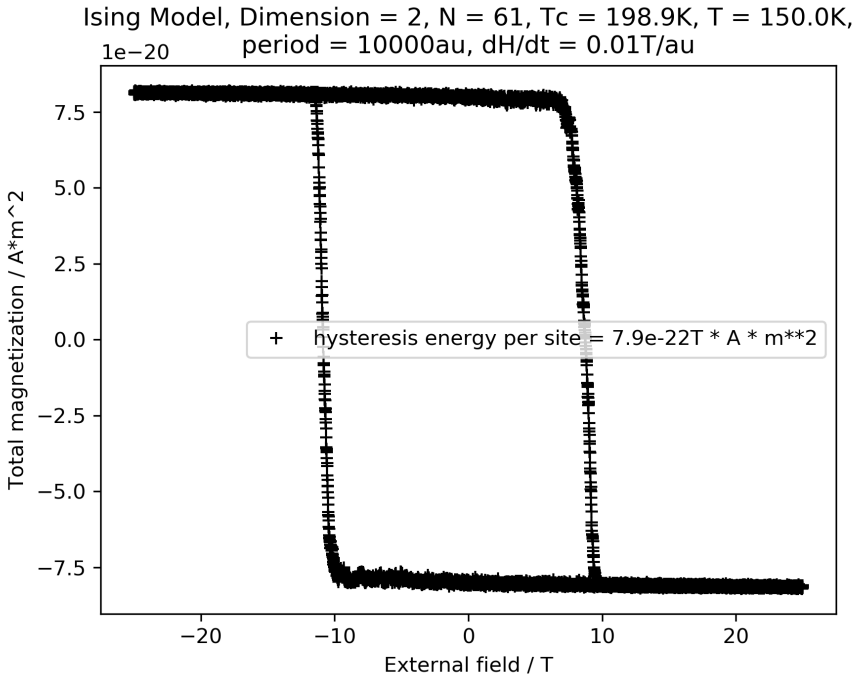

FIG. 23: Finite size scaling of Curie temperature against system size.

At $150K$ our system exhibits hysteresis behaviour 23.

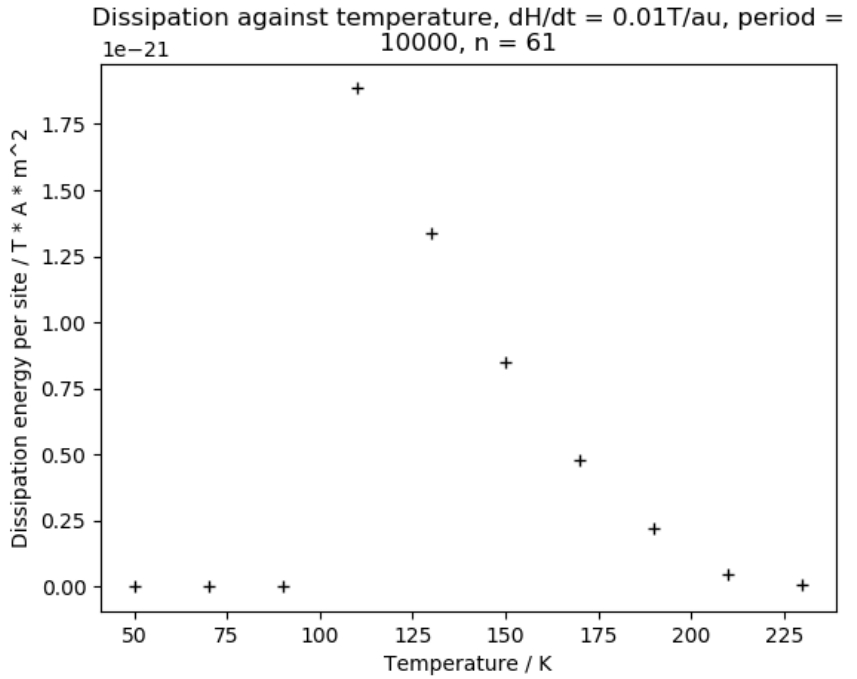

FIG. 24: Dissipation energy per site against temperature.

In disordered phase, little dissipation is observed for the non-ferromagnetic system. In ordered phase, dissipation disappears below $110K$ 24 as the external field fails to flip the system. Ferromagnetic strength decreases with temperature above $110K$.

IX. DISCUSSIONS

The Checkerboard algorithm has enabled repeated calculation of large systems compared to the Metropolis algorithm. Near Curie point performance can still be improved using the Wolfson clustering. Temperature dependence of coercive field and saturation field can be studied in the future.

X. SUMMARY

Checkerboard algorithm [1] proves more efficient than the Metropolis algorithm as the system increase in size 8. Equilibrium time, correlation time, equilibrium magnetization, finite-size scaling and hysteresis behaviours corroborated theoretical analysis.

This paper has 8286 words, with 4983 in the appendix, 59 in the summary and approximately 240 (10*24) in figure captions.

XI. APPENDIX

```
# common computing and statistical packages
import numpy as np
import numpy.linalg as linalg
import scipy as sp
import scipy.stats
import statsmodels.tsa.stattools
import random
import time
import copy

# local convolution for local interaction energy
from scipy.ndimage import convolve, generate_binary_structure
from scipy.optimize import curve_fit

# error estimators
from astropy.stats import jackknife_stats
from astropy.stats import bootstrap

# Gaussian clustering
from itertools import cycle
from sklearn import mixture

import os  # writing to path
import sigfig  # rounding for data in plot title
from textwrap3 import wrap

import matplotlib as mpl
mpl.use('Agg')
from matplotlib import pyplot as plt

# ------------------------------------------------------------------------------

# SI Units ommited
```

```
# @jitclass(spec)
class Ising:
    # iron from https://www.southampton.ac.uk/~rpb/thesis/node18.html
    def __init__(self, name="none", N=100, H=0, T=273.15, D=2, J=1.21 * np.power(10.0, -21),
        self.removeUnderflow = np.power(10, 0)
        self.j = J  # / self.removeUnderflow  # 'numerical' coupling constant
        self.h = H  # external field strength
        self.n = N  # lattice size
        self.m = M  # single spin magnetic moment
        self.k = K  # boltzman constant
        self.t = T  # / self.removeUnderflow # temperature in kelvins
        self.d = D  # system dimension
        self.tc = 2*J/(K*np.arcsinh(1))  # theoretical tc for 2d by onsanger
        self.timeStep = 0  # initialize timestep marker
        # figure sizes
        self.figureScale = 2
        self.figureDpi = 300 # matplotlib defaults to dpi=100
        # in inches
        self.figureHeight = 4.8
        self.figureWidth = 6.4
        # working
        self.kernel = generate_binary_structure(self.d, 1)
        self.ground = np.full(np.repeat(self.n, self.d), 0)
        # storage of system physical property time series: time stamp, total magnetization, t
        self.systemDataTimeSeries = [[], [], []]
        # storage of blocksize, measured magnetization, variance of measured magnetization, v
        self.correlationTimeEstimates = [[], [], []]
        self.binderCumulant = None
        # normalized magnetization auto-covariance
        self.normalizedMagnetizationAutocovariance = None
        # Time needed to reach steady state from initial state, initially set to zero
        # will be updated if systems takes time t to move into equilibrium

        # mean magnetization and its error from a single measurement, returned as a 2 element
        self.meanMagnetizationWithError = []

        self.equilibriumTime = equilibriumTime
        self.numberOfMeasurements = numberOfMeasurements
        # correlation time initially set to 2
        # can be precomputed then provided
        # Warning: THIS IS A REAL
        self.correlationTime = correlationTime
        # this cannot be provided
        self.specificHeatCapacityPerSite = None
        # plotting colors
        self.colors_ = cycle(mpl.colors.TABLEAU_COLORS.keys())

        # define the system with or without initial values
        spins = [1, -1]
        if randomFill:
            self.system = np.random.choice(spins, tuple(np.repeat(self.n, self.d)))
        else:
            # dangerously, for future importing of existing system
            # currently randomly chooses all up or all down
            self.system = np.full(tuple(np.repeat(self.n, self.d)), np.choose(1, [1, -1]))
        self.system = np.asarray(self.system)
        choices = list(range(self.n**self.d))
```

```
        # warning: optimization only works in odd sized dimensions!!!
        zerothSites = choices[0::4]
        firstSites = choices[1::4]
        secondSites = choices[2::4]
        thirdSites = choices[3::4]
        # notice we still have the difficulty of flipping large domains with smooth boundarie


        # dimension index
        self.dimensions = range(self.d)

        # zerothLattice
        self.zerothLattice = np.full(tuple(np.repeat(self.n, self.d)), False)
        for choice in zerothSites:
            self.zerothLattice[tuple([int(np.floor(choice % self.n**(x+1) / self.n**x)) for x

        # firstLattice
        self.firstLattice = np.full(tuple(np.repeat(self.n, self.d)), False)
        for choice in firstSites:
            self.firstLattice[tuple([int(np.floor(choice % self.n**(x+1) / self.n**x)) for x i

        # secondLattice
        self.secondLattice = np.full(tuple(np.repeat(self.n, self.d)), False)
        for choice in secondSites:
            self.secondLattice[tuple([int(np.floor(choice % self.n**(x+1) / self.n**x)) for x

        # thirdLattice
        self.thirdLattice = np.full(tuple(np.repeat(self.n, self.d)), False)
        for choice in thirdSites:
            self.thirdLattice[tuple([int(np.floor(choice % self.n**(x+1) / self.n**x)) for x i

        # in each step randomly update all sublattices
        self.sublattices = np.array([self.zerothLattice, self.firstLattice, self.secondLattice

        # self.evenLattice = np.invert(copy.deepcopy(self.oddLattice))
        # iniitalize initial energies using initial state
        self.updateEnergies()

        # save initial system data at time stamp=0
        self.systemDataTimeSeries[0].append(self.timeStep)
        self.systemDataTimeSeries[1].append(self.totalMagnetization())
        # care: coordination number normalization when accounting for total energy
        self.systemDataTimeSeries[2].append(np.sum(self.interactionEnergies) / 2)

        self.stationaryMagnetization = None

    def totalMagnetization(self):
        return self.m * np.sum(self.system)

    # Warning: estimate the equilibrium time before calculating binderCumulent
    def estimateBinderCumulant(self):
        data = copy.deepcopy(self.systemDataTimeSeries)
        # crop transients
        if not self.equilibriumTime == 0:
            data[0] = data[0][slice(int(self.equilibriumTime * 2), self.timeStep)]
            data[1] = data[1][slice(int(self.equilibriumTime * 2), self.timeStep)]
            data[2] = data[2][slice(int(self.equilibriumTime * 2), self.timeStep)]
        meanToTheFourth = np.mean(np.power(data[1], 4.0))
```

```
        meanToTheSecond = np.mean(np.power(data[1], 2.0))
        self.binderCumulant = 1 - meanToTheFourth / (3 * meanToTheSecond ** 2)

    def localEnergy(self, coords):  # periodic bc
        # coords a list contain d integer indices to specify the ising lattice in d dimension
        sumOfNeighbours = 0
        for i in range(len(coords)):  # traverse in f/b directions
            coordsCopy = copy.deepcopy(coords)  # deep copy by default
            coordsCopy[i] = (coords[i] + 1) % self.n
            sumOfNeighbours += self.system[tuple([[x] for x in coordsCopy])][0]
            coordsCopy[i] = (coordsCopy[i] - 2) % self.n
            sumOfNeighbours += self.system[tuple([[x] for x in coordsCopy])][0]

        coords = tuple([[x] for x in coords])
        return (- self.j * (self.system[coords]) * sumOfNeighbours +
                self.m * self.h * self.system[coords])[0]
        # coupling energy + external field interaction

    # function for parallel update of interaction energies
    def updateEnergies(self):
        self.interactionEnergies = \
                (-self.j) * (convolve(self.system, self.kernel, mode='wrap') - self.system) *
                self.m * self.h * self.system

    # currently deprecated
    def flip(self, coords):
        energy = self.localEnergy(coords)
        # print(energy)
        coords = tuple([[x] for x in coords])
        if energy >= 0:  # flip
            self.system[coords] *= -1
        else:
            boltzmanFactor = np.exp(2*energy/(self.k * self.t))
            # p = random.uniform(0, 1)
            if random.randint(0, 1) < boltzmanFactor: self.system[coords] = -self.system[coor

    def visualizeMagnetization(self, path="noPath.png", hyperplane=None):
        # plots the total magnetization with time
        plt.close()
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        # fig = plt.figure()
        plt.plot(self.systemDataTimeSeries[0], self.systemDataTimeSeries[1], "+k")
        plt.axvline(x=self.equilibriumTime)
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", N = "+str(self.n)+
        plt.xlabel("Time steps / a.u.")
        plt.ylabel("Total magnetization / Am^2")
        return fig

    def visualizeTotalEnergy(self, path="noPath.png", hyperplane=None):
        # plots the total magnetization with time
        plt.close()
        # fig = plt.figure()
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        plt.plot(self.systemDataTimeSeries[0], self.systemDataTimeSeries[2], "+k")
        plt.axvline(x=self.equilibriumTime)
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", N = "+str(self.n)+
        plt.xlabel("Time steps / a.u.")
        plt.ylabel("Total energy / J")
```

```
        return fig

    def meanStationaryMagnetization(self, path="noPath.png", hyperplane=None):
        # calculate mean stationary magnetization and its variance
        # return a two element list

        # remove the transient data
        if not self.equilibriumTime == 0:
            steadyStateDataTimeSeries = copy.deepcopy(self.systemDataTimeSeries[1][slice(int(
        else:
            steadyStateDataTimeSeries = copy.deepcopy(self.systemDataTimeSeries[1])
        blockSize = int(self.correlationTime) + 1
        numberOfMeasurements = self.numberOfMeasurements

        # measure in blocks of size the correlation length
        averages = np.array([])
        for i in range(numberOfMeasurements):
            if (i + 1) * blockSize < len(steadyStateDataTimeSeries): # prevent running off in
                averages = np.append(averages, [np.mean(steadyStateDataTimeSeries[slice(i * b
            else:
                break

        self.stationaryMagnetization = [np.mean(averages), np.sqrt(np.var(averages) / (self.nu
        return self.stationaryMagnetization

    def visualizeMagnetizationAutocovariance(self, path="noPath.png", hyperplane=None):
        plt.close()
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        data = copy.deepcopy(self.systemDataTimeSeries)
        # crop transients
        if not self.equilibriumTime == 0:
            data[0] = data[0][slice(int(self.equilibriumTime * 2), self.timeStep)]
            data[1] = data[1][slice(int(self.equilibriumTime * 2), self.timeStep)]
            data[2] = data[2][slice(int(self.equilibriumTime * 2), self.timeStep)]

        # returns an auto-covariance plot
        magnetizationAutocovariance = statsmodels.tsa.stattools.acovf(data[1], demean=True, ft
        # with non-equilibrium cropped
        normalizedMagnetizationAutocovariance = magnetizationAutocovariance/magnetizationAuto
        # save current autocovariance
        self.normalizedMagnetizationAutocovariance = normalizedMagnetizationAutocovariance
        # fig = plt.figure()
        plt.plot(data[0], normalizedMagnetizationAutocovariance, "+k")
        plt.axvline(x=self.equilibriumTime)
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", N = "+str(self.n)+
        plt.xlabel("Time steps / a.u.")
        plt.ylabel("Auto-covariance of magnetization")
        return fig

    # warning, only estimate the correlation time after computing the auto-covariance time se
    def estimateCorrelationTime(self):
        cropLength = 0
        cropFinished = False
        # case where correlation time is not estimated is not handled here
        for i in range(self.timeStep - int(self.equilibriumTime * 2)):
            if cropFinished:
                break
            if self.normalizedMagnetizationAutocovariance[i] <= np.exp(-2.0):
```

```
                cropLength = i
                cropFinished = True

        # this is a more sophisticated method by fitting the autocovariance curve
        """
        croppedAutocovariance = copy.deepcopy(self.normalizedMagnetizationAutocovariance[:crop
        time = np.arange(cropLength)

        # print(croppedAutocovariance)
        logCroppedAutocovariance = np.log(croppedAutocovariance)
        # logTime = np.log(time)
        b, m = np.polyfit(time, logCroppedAutocovariance, 1)
        self.correlationTime = np.absolute(1 / m)
        """
        self.correlationTime = cropLength

        # successfully estimated correlation time
        return True

    def estimateSpecificHeatCapacityPerSite(self):
        data = copy.deepcopy(self.systemDataTimeSeries)
        # crop transients
        if not self.equilibriumTime == 0:
            data[0] = data[0][slice(int(self.equilibriumTime * 2), self.timeStep)]
            data[1] = data[1][slice(int(self.equilibriumTime * 2), self.timeStep)]
            data[2] = data[2][slice(int(self.equilibriumTime * 2), self.timeStep)]
        energyTimeSeriesPerSite = data[2] / np.power(self.n, self.d)
        # fluctuation dissipation theorem
        specificHeatCapacityPerSite = np.sqrt((np.var(energyTimeSeriesPerSite))/(self.k * self
        self.specificHeatCapacityPerSite = specificHeatCapacityPerSite
        return True

    # data cleaning function that demean and normalize data using its absolute magnitude
    def demeanNormalize(self, ndarray):
        ndarray = copy.deepcopy(ndarray)
        ndarray = ndarray / np.mean(np.absolute(ndarray))
        ndarray = ndarray - np.mean(ndarray)
        return ndarray

    # helper cluster data plotter from scikit-learn, note plots at 1, 2, n sub graph for comp
    def plotClustersEstimateEquilibriumTime(self, X, model, subplot):
        Y_ = model.predict(X)
        covariances = model.covariances_
        means = model.means_
        for i, (mean, covar, color) in enumerate(zip(
            means, covariances, self.colors_)):
        # for i, (mean, covar) in enumerate(zip(
        #     means, covariances)):
            v, w = linalg.eigh(covar)
            v = 2. * np.sqrt(2.) * np.sqrt(v)
            u = w[0] / linalg.norm(w[0])
            # as the DP will not use every component it has access to
            # unless it needs it, we shouldn't plot the redundant
            # components.
            if not np.any(Y_ == i):
                continue
            subplot.scatter(X[Y_ == i, 0], X[Y_ == i, 1], .8, color=color)
```

```
            # Plot an ellipse to show the Gaussian component
            angle = np.arctan(u[1] / u[0])
            angle = 180. * angle / np.pi  # convert to degrees
            ell = mpl.patches.Ellipse(mean, v[0], v[1], 180. + angle, color=color)
            ell.set_clip_box(subplot.bbox)
            ell.set_alpha(0.5)
            subplot.add_artist(ell)

        # identify number of steps to equilibrium, if any steps needed at all
        modelLabels = Y_
        labels, counts = np.unique(modelLabels[modelLabels >= 0], return_counts=True)
        if np.absolute(counts[0] - counts[1]) / self.timeStep < 0.5:
            self.equilibriumTime = 0
            print("Either already in equilibrium state or system has not reached equilibrium..
        else:
            self.equilibriumTime = np.amin(counts)
            print("Time to equilibrium: " + str(self.equilibriumTime))

    def visualizeMagnetizationPhaseSpace(self, path="noPath.png", hyperplane=None):
        # plots the total magnetization with its time
        plt.close()
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale * 2, self.figureHeight *
        # import and reshape data
        # scikit-learn only takes columns of attributes
        magnetization = self.demeanNormalize(self.systemDataTimeSeries[1])
        magnetization = np.reshape(magnetization, (magnetization.size, 1))
        magnetizationGradient = self.demeanNormalize(np.gradient(self.systemDataTimeSeries[1])
        magnetizationGradient = np.reshape(magnetizationGradient, (magnetizationGradient.size

        # original data
        ax = fig.add_subplot(1, 2, 1)
        ax.plot(magnetization, magnetizationGradient, "+k")
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", N = "+str(self.n)+
        plt.xlabel("demeaned and magnitude normalized Magnetization / a.u.")
        plt.ylabel("d(demeaned and magnitude normalized Magnetization)/dt / a.u.")

        # clustering using scikit.learn
        X = np.hstack((magnetization, magnetizationGradient))
        # model = Birch(threshold=0.5, n_clusters=2)
        # model = KMeans(n_clusters=2)
        model = mixture.GaussianMixture(n_components=2, covariance_type='full')
        model.fit(X)
        ax = fig.add_subplot(1, 2, 2)
        self.plotClustersEstimateEquilibriumTime(X, model, ax)
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", N = "+str(self.n)+
        plt.xlabel("demeaned and magnitude normalized Magnetization / a.u.")
        plt.ylabel("d(demeaned and magnitude normalized Magnetization)/dt / a.u.")
        return fig

    def visualizeTwoDGrid(self, path="noPath.png", hyperplane=None):
        # safety measure: close all plots
        plt.close()
        # hyperplane should be an integer indexing list
        cmap = mpl.colors.ListedColormap(['white', 'black'])
        bounds = [-1, 0, 1]
        norm = mpl.colors.BoundaryNorm(bounds, cmap.N)
        # data reshape
        data = copy.deepcopy(self.system[hyperplane])
```

```
        if data.shape == tuple([self.n, self.n]):
            pass
        else:
            data = data[:, :, 0]
        # plot
        fig, axes = plt.subplots()
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        axes = fig.add_subplot(111)
        img = axes.imshow(data, interpolation='nearest', cmap=cmap, norm=norm, animated=True)

        plt.colorbar(img, cmap=cmap, norm=norm, boundaries=bounds, ticks=[-1, 0, 1])
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", N = "+str(self.n)+
        return fig

    # warning: do not run on its own
    # this is a helper function for stepForward()
    def updateSublattice(self, sublattice):
        boltzmanFactor = np.exp(2 * self.interactionEnergies / (self.k * self.t))
        evenDist = np.random.uniform(0, 1, size=np.repeat(self.n, self.d))
        temp1 = np.greater(self.interactionEnergies, self.ground)
        temp2 = np.greater(boltzmanFactor, evenDist)
        criteria = np.logical_and(sublattice, np.logical_or(temp1, temp2))
        self.system = np.where(criteria, -self.system, self.system)
        self.updateEnergies()

    def stepForward(self):
        # stepping through the lattice and update randomly
        # improved method: divide into two sub-lattices and vectorize for each sub lattice to
        # note that a site cannot be updated twice within a single step, and two neighbouring
        self.timeStep = self.timeStep + 1

        np.random.shuffle(self.sublattices)
        for sublattice in self.sublattices:
            self.updateSublattice(sublattice)

        # record system data
        self.systemDataTimeSeries[0].append(self.timeStep)
        self.systemDataTimeSeries[1].append(self.totalMagnetization())
        # care: coordination number normalization when accounting for total energy to avoid d
        self.systemDataTimeSeries[2].append(np.sum(self.interactionEnergies) / 2)


# ------------------------------------------------------------------------------------------
# calculate the time evolution of a 2 D ising model system given the predefined parameters


# working now: investigate the hysteresis effect in a 2 d system
# the hysteresis by cycling h
# measure the remnant field
# measure the external field needed for total negating of the field
# measure the hysteresis loop energy, notice need to add impurities if you want a considerable

# measurement class to allow taking measurements across multiple systems
# @jitclass(spec)
class InquireIsing:
    def __init__(self, name="none", N=100, H=0, T=273.15, D=2, J=1.21 * np.power(10.0, -21),
        self.removeUnderflow = np.power(10, 0)
        self.j = J  # / self.removeUnderflow  # 'numerical' coupling constant
```

```
        self.h = H  # external field strength
        self.n = N  # lattice size
        self.m = M  # single spin magnetic moment
        self.k = K  # boltzman constant
        self.t = T  # / self.removeUnderflow # temperature in kelvins
        self.d = D  # system dimension
        self.tc = 2*J/(K*np.arcsinh(1))  # theoretical tc for 2d by onsanger
        self.randomFill = randomFill
        self.steps = steps
        self.numberOfMeasurements = numberOfMeasurements
        self.dataPoints = dataPoints
        self.lowerTemperature = lowerTemperature
        self.deltaTemperature = deltaTemperature
        # figure sizes
        self.figureScale = 1
        self.figureDpi = 300 # matplotlib defaults to dpi=100
        # in inches
        self.figureHeight = 4.8
        self.figureWidth = 6.4
        # Time needed to reach steady state from initial state, initially set to zero
        # will be updated if systems takes time t to move into equilibrium
        self.equilibriumTime = equilibriumTime
        self.name = name
        # correlation time initially set to 2
        # can be inputed if precomputed
        self.correlationTime = correlationTime
        # plotting colors
        # self.colors_ = cycle(mpl.colors.TABLEAU_COLORS.keys())

        # data
        # size of block, mean variance in mean steady state magnetization, variance in the va
        # self.blockSizeAndVarianceInMeanMagnetization = [[], [], []]
        self.meanStationaryMagnetizationAgainstTemperature = None
        self.equilibriumTimeAgainstTemperature = None
        self.correlationTimeAgainstTemperature = None
        self.binderCumulantAgainstTemperature = None
        self.specificHeatCapacityPerSiteAgainstTemperature = None

    def visualizeBinderCumulant(self):
        # plots the average magnetization vs number of blocks used for measurement
        # when the variance starts to grow wildly, we choose the largest block size with non-
        plt.close()
        lowerTemperature = self.lowerTemperature
        deltaTemperature = self.deltaTemperature
        numberOfMeasurements = self.numberOfMeasurements
        attempts = self.dataPoints
        # temperature and equilibriumTime and error
        data = [[], [], []]
        for a in range(attempts):
            temperature = lowerTemperature + a * deltaTemperature
            data[0].append(temperature)
            binderCumulants = []
            for j in range(numberOfMeasurements):
                tempSys = Ising(name=self.name, N=self.n, H=self.h, T=temperature, D=self.d, J
                for i in range(self.steps):
                    tempSys.stepForward()
                # estimate equilibrium time needed
                tempSys.visualizeMagnetizationPhaseSpace()
```

```
            tempSys.estimateBinderCumulant()
            binderCumulants.append(tempSys.binderCumulant)
            # [meanMagnetization, errorMagnetization] = tempSys.meanStationaryMagnetizatio
            # errors.append(errorMagnetization)
        data[1].append(np.mean(binderCumulants))

        # three ways to estimate errors
        _input = np.array(binderCumulants)
        _statistics = np.mean

        sem = scipy.stats.sem(_input) # only works for mean
        jackknife_estimate, bias, stderr, conf_interval = jackknife_stats(np.array(_input)
        bootstrap_estimate = bootstrap(_input, bootfunc=_statistics)

        data[2].append(jackknife_estimate)

    # update stored data
    self.binderCumulantAginstTemperature = data
    # fig = plt.figure()
    fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
    # print(self.blockSizeAndVarianceInMeanMagnetization)
    plt.errorbar(self.binderCumulantAgainstTemperature[0], self.binderCumulantAgainstTemp
    # plt.axvline(x=self.equilibriumTime)
    plt.title("\n".join(wrap("Ising␣Model,␣Dimension␣=␣"+str(tempSys.d)+",␣N␣=␣"+str(temp
    plt.xlabel("Temperature␣/␣K")
    plt.ylabel("Binder␣Cumulant␣/␣1")
    return fig



def visualizeEquilibriumTime(self):
    # plots the average magnetization vs number of blocks used for measurement
    # when the variance starts to grow wildly, we choose the largest block size with non-
    plt.close()
    lowerTemperature = self.lowerTemperature
    deltaTemperature = self.deltaTemperature
    numberOfMeasurements = self.numberOfMeasurements
    attempts = self.dataPoints
    # temperature and equilibriumTime and error
    data = [[], [], []]
    for a in range(attempts):
        temperature = lowerTemperature + a * deltaTemperature
        data[0].append(temperature)
        # self.blockSizeAndVarianceInMeanMagnetization[0].append(blockSize)
        equilibriumTimes = []
        for j in range(numberOfMeasurements):
            tempSys = Ising(name=self.name, N=self.n, H=self.h, T=temperature, D=self.d, J
            for i in range(self.steps):
                tempSys.stepForward()
            # estimate equilibrium time needed
            tempSys.visualizeMagnetizationPhaseSpace()
            equilibriumTimes.append(tempSys.equilibriumTime)
            # [meanMagnetization, errorMagnetization] = tempSys.meanStationaryMagnetizatio
            # errors.append(errorMagnetization)
        data[1].append(np.mean(equilibriumTimes))
        data[2].append(np.sqrt(np.var(equilibriumTimes)))
    # update stored data
    self.equilibriumTimeAginstTemperature = data
```

```
        # fig = plt.figure()
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        # print(self.blockSizeAndVarianceInMeanMagnetization)
        plt.errorbar(self.equilibriumTimeAgainstTemperature[0], self.equilibriumTimeAgainstTem
        # plt.axvline(x=self.equilibriumTime)
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(tempSys.d)+", N = "+str(temp
        plt.xlabel("Temperature / K")
        plt.ylabel("Equilibrium Time / a.u.")
        return fig

    def visualizeStationaryMagnetization(self):
        # plots the average magnetization vs number of blocks used for measurement
        # when the variance starts to grow wildly, we choose the largest block size with non-
        plt.close()
        lowerTemperature = self.lowerTemperature
        deltaTemperature = self.deltaTemperature
        numberOfMeasurements = self.numberOfMeasurements
        dataPoints = self.dataPoints
        # temperature and mean magnetization and error
        data = [[], [], []]
        for a in range(dataPoints):
            temperature = lowerTemperature + a * deltaTemperature

            # compute the correlation time under current temp
            # !!! to be finished !!!

            data[0].append(temperature)
            # self.blockSizeAndVarianceInMeanMagnetization[0].append(blockSize)
            tempSys = Ising(name=self.name, N=self.n, H=self.h, T=temperature, D=self.d, J=se
            for step in range(self.steps):
                tempSys.stepForward()

            # necessary facilitating computations
            tempSys.visualizeMagnetizationPhaseSpace()
            tempSys.visualizeMagnetizationAutocovariance()
            tempSys.estimateCorrelationTime()
            print(tempSys.correlationTime)

            [meanMag, error] = tempSys.meanStationaryMagnetization()
            data[1].append(meanMag)
            data[2].append(error)
        # update stored data
        self.meanStationaryMagnetizationAgainstTemperature = data
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        plt.errorbar(self.meanStationaryMagnetizationAgainstTemperature[0], self.meanStationar
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", N = "+str(self.n)+
        plt.xlabel("Temperature / K")
        plt.ylabel("Stationary Magnetization / Am^2")
        return fig

    # helper function returning correlation time for a specific temperature, keeping all othe
    # used in function visualizeCorrelationTime
    def correlationTimeByTemperature(self, T=200):
        correlationTimes = []
        j = 0
        while j < self.numberOfMeasurements:
            # use specified temperature
            tempSys = Ising(name=self.name, N=self.n, H=self.h, T=T, D=self.d, J=self.j, rand
```

```
            for step in range(self.steps):
                tempSys.stepForward()
            # estimate equilibrium time
            tempSys.visualizeMagnetizationPhaseSpace()
            # compute auto correlation time series
            tempSys.visualizeMagnetizationAutocovariance()
            # estimate correlation time
            tempSys.estimateCorrelationTime()
            j = j + 1
            correlationTimes.append(tempSys.correlationTime)
        return [np.mean(correlationTimes), np.sqrt(np.var(correlationTimes))]

    def visualizeCorrelationTime(self, n=31, delta=10, size=3):
        # plots the average magnetization vs number of blocks used for measurement
        # when the variance starts to grow wildly, we choose the largest block size with non-
        plt.close()
        lowerTemperature = self.lowerTemperature
        deltaTemperature = self.deltaTemperature
        numberOfMeasurements = self.numberOfMeasurements
        dataPoints = self.dataPoints

        # fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight *
        datas = []
        for i in range(size):
            # temperature and mean magnetization and error
            data = [[], [], []]
            _n = n + i * delta
            self.n = _n
            for a in range(dataPoints):
                temperature = lowerTemperature + a * deltaTemperature + i * 0.1 * deltaTemper
                data[0].append(temperature)
                # offset the data points for visual purposes
                correlationTime = self.correlationTimeByTemperature(temperature)
                data[1].append(correlationTime[0])
                data[2].append(correlationTime[1])
            # update stored data
            self.correlationTimeAgainstTemperature = data
            # fig = plt.figure()
            datas.append(copy.deepcopy(data))
            # for j in range(size)
        # eee
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        # kept for record the followuing line, do not remove plz
        # plt.errorbar(self.correlationTimeAgainstTemperature[0], self.correlationTimeAgainstT
        for i in range(size):
            plt.errorbar(datas[i][0], datas[i][1], yerr=datas[i][2], fmt="+", label="n = "+str

        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", Tc = "+str(sigfig
        plt.legend()
        plt.xlabel("Temperature / K")
        plt.ylabel("Correlation Time / a. u.")
        return fig

    def specificHeatCapacityPerSiteByTemperature(self, T=200):
        specificHeatCapacities = []
        j = 0
        while j < self.numberOfMeasurements:
            # use specified temperature
```

```
            tempSys = Ising(name=self.name, N=self.n, H=self.h, T=T, D=self.d, J=self.j, rand
            for step in range(self.steps):
                tempSys.stepForward()
            # estimate equilibrium time
            tempSys.visualizeMagnetizationPhaseSpace()
            tempSys.estimateSpecificHeatCapacityPerSite()
            j = j + 1
            # halted
            specificHeatCapacities.append(tempSys.specificHeatCapacityPerSite)
        return [np.mean(specificHeatCapacities), np.sqrt(np.var(specificHeatCapacities))]

    def visualizeSpecificHeatiCapacityPerSite(self):
        plt.close()
        lowerTemperature = self.lowerTemperature
        deltaTemperature = self.deltaTemperature
        dataPoints = self.dataPoints
        # temperature and mean magnetization and error
        data = [[], [], []]
        for a in range(dataPoints):
            temperature = lowerTemperature + a * deltaTemperature
            data[0].append(temperature)
            specificHeatCapacity = self.specificHeatCapacityPerSiteByTemperature(temperature)
            data[1].append(specificHeatCapacity[0])
            data[2].append(specificHeatCapacity[1])
        # update stored data
        self.specificHeatCapacityPerSiteAgainstTemperature = data
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        plt.errorbar(self.specificHeatCapacityPerSiteAgainstTemperature[0], self.specificHeatC
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", N = "+str(self.n)+
        plt.xlabel("Temperature / K")
        plt.ylabel("Specific Heat Capacity per site / J * K ^ -1")
        return fig

    def curieTemp(self, n=11, tLow=190, accuracy=2, size=4):
        # curie temp, error
        data = []

        self.n = n
        specific_heats_data = [[], []]
        for i in range(size):
            temperature = tLow + i * accuracy
            specific_heat = self.specificHeatCapacityPerSiteByTemperature(temperature)[0]
            specific_heats_data[0].append(temperature)
            specific_heats_data[1].append(specific_heat)
        a = specific_heats_data[1]
        data = [specific_heats_data[0][a.index(max(a))], accuracy]
        return data

    def visualizeCurieTemperatureAgainstTime(self, n=11, delta=4, length_eles=3, tLow=199, ac
        # size, curie temp, error
        data = [[], [], []]
        # print(self.k)
        for i in range(length_eles):
            _n = n + delta * i
            a = self.curieTemp(_n, tLow, accuracy, temp_eles)
            data[0].append(_n)
            data[1].append(a[0])
            data[2].append(a[1])
```

```
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        # temperature in J/k_B units plotted
        data[1] = data[1] / (self.j/self.k)
        data[2] = data[2] / (self.j/self.k)
        plt.errorbar(data[0], data[1], yerr=data[2], fmt="+k")
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", Tc = "+str(sigfig
        plt.xlabel("Lattice Length ")
        plt.ylabel("Curie Temperature / J/k_B")
        return fig

    # function to fit, look for tc, n is an array
    def f(self, n, tc, a, nu):
        return tc + a * np.power(n, -1/nu)

    def finiteSizeScaling(self, n=11, delta=4, length_eles=3, tLow=199, accuracy=0.6, temp_el
        # size, curie temp, error
        data = [[], [], []]
        for i in range(length_eles):
            _n = n + delta * i
            a = self.curieTemp(_n, tLow, accuracy, temp_eles)
            data[0].append(_n)
            data[1].append(a[0])
            data[2].append(a[1])
        fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se
        # temperature in J/k_B units plotted
        data[1] = data[1] / (self.j/self.k)
        data[2] = data[2] / (self.j/self.k)
        func = self.f
        xdata = data[0]
        ydata = data[1]
        ysigma = data[2]
        popt, pcov = curve_fit(func, xdata, ydata, sigma=ysigma, absolute_sigma=True)
        perr = np.sqrt(np.diag(pcov))
        tc = [popt[0], perr[0]]
        a = [popt[1], perr[1]]
        nu = [popt[2], perr[2]]
        label = "Tc = "+str(tc[0])+" +- " + str(tc[1])
        plt.errorbar(data[0], data[1], yerr=data[2], fmt="+k", label=label)
        plt.legend()
        plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", Tc = "+str(sigfig
        plt.xlabel("Lattice Length")
        plt.ylabel("Curie Temperature / J/k_B")
        return fig

    def hysteresis(self, temperature=150, delta_H=0.01, period=10000):
        tempSys = Ising(name=self.name, N=self.n, H=self.h, T=temperature, D=self.d, J=self.j
        # external field in tesla, internal magnetization
        data = [[], []]
        for i in range(int(period/4)):
            tempSys.h = tempSys.h + (delta_H)
            data[0].append(tempSys.h)
            tempSys.stepForward()
        # calculate over more cycles to remove the initial moving influence

        cycles = 3
        for j in range(cycles):
            for i in range(int(period/4)):
                tempSys.h = tempSys.h - (delta_H)
```

```
            data[0].append(tempSys.h)
            tempSys.stepForward()
        for i in range(int(period/4)):
            tempSys.h = tempSys.h - (delta_H)
            data[0].append(tempSys.h)
            tempSys.stepForward()
        for i in range(int(period/4)):
            tempSys.h = tempSys.h + (delta_H)
            data[0].append(tempSys.h)
            tempSys.stepForward()
        for i in range(int(period/4)):
            tempSys.h = tempSys.h + (delta_H)
            data[0].append(tempSys.h)
            tempSys.stepForward()

    data[1] = tempSys.systemDataTimeSeries[1]
    xoffset = int(period/4)
    yoffset = int(period/4 + 1)
    # points = np.column_stack((data[0][xoffset], data[1][yoffset]))
    # hull = ConvexHull(points)
    switching_energy = 0
    t = int(period/4)
    for j in range(cycles):
        for i in range(int(period/4)):
            t = t + 1
            switching_energy += delta_H * data[1][t] * (-1)
        for i in range(int(period/4)):
            t = t + 1
            switching_energy += delta_H * data[1][t] * (-1)
        for i in range(int(period/4)):
            t = t + 1
            switching_energy += delta_H * data[1][t] * (+1)
        for i in range(int(period/4)):
            t = t + 1
            switching_energy += delta_H * data[1][t] * (+1)


    # per site
    switching_energy = switching_energy / self.n**2 / cycles
    fig = plt.figure(figsize=(self.figureWidth * self.figureScale, self.figureHeight * se

    # previous plotting
    xend = period + xoffset
    yend = period + yoffset
    plt.plot(data[0][xoffset:xend], data[1][yoffset:yend], "+k", label = "dissipation ener

    plt.title("\n".join(wrap("Ising Model, Dimension = "+str(self.d)+", N = "+str(self.n)+
    plt.legend()
    plt.xlabel("External field / T")
    plt.ylabel("Total magnetization / A*m^2")
    return [temperature, switching_energy]
```

---

* Many thanks to my parents for supporting my various pursuits.

† lm766@cam.ac.uk

[1] B. Block, P. Virnau, and T. Preis, Computer Physics Communications **181**, 1549 (2010).

[2] N. D. Mermin and H. Wagner, Phys. Rev. Lett. **17**, 1133 (1966).
[3] L. Onsager, Phys. Rev. **65**, 117 (1944).
[4] A. Liccardo and A. Fierro, PLOS ONE **8**, 1 (2013).
[5] W. Selke, European Physical Journal B **51**, 223 (2006), arXiv:0603411 [cond-mat].
[6] R. Boardman, *Computer simulation studies of magnetic nanostructures*, Ph.D. thesis, University of Southampton (2005).
[7] D. Buscher, *Part II Computational Physics*, Tech. Rep. (2020).