# 2D Ising Model Of A Ferromagnet

Candidate Number: 6950X

***Abstract*—In this project, physical properties of a 2D Ising model of a ferromagnet will be investigated using Monte Carlo simulation with importance sampling, based on the Metropolis-Hastings algorithm. We will look at how temperature and lattice size affect the magnetisation, energy, heat capacity, and magnetic susceptibility of the system, and how these infer a phase transition in the system. Finite-size scaling will also be discussed.**

## I. Introduction

The main aim of this project is to explore the aspects of computational physics, especially the Monte Carlo method in the context of a 2D Ising model of a ferromagnet. Background theory on this stochastic approach and the Ising model is presented, and the actual implementation of the Metropolis-Hastings algorithm will also be given.

An Ising model is introduced and used to investigate the behaviour of a two dimensional ferromagnet while varying different parameters such as the lattice size and temperatures. Relevant thermodynamic variables are sampled and presented, demonstrating the phase transition process of a ferromagnet at a certain critical temperature. How these observables evolve and fluctuate with time will also be quantified. Last but not least, a brief finite-size scaling analysis is carried out to determine the critical exponent of magnetisation as well as the critical temperature. All these results are compared to their theoretical counterparts to check the validity of this numerical method.

The Python script that is used can be found in the appendix, and is freely available for use and modification under the General Public License.

## II. Background

Ferromagnetism is an intrinsically many-body quantum phenomenon that would require quantum rules of spin and angular momentum. The Ising model allows us to simplify the complexity associated with the quantum mechanical nature of the problem while still retaining its essential physics and allowing a good conceptual understanding of the phenomenon.

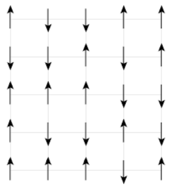

Fig. 1: Illustration of a 2D Ising lattice

Consider a square set of lattice sites of size $N \times N$, each site accommodates a spin $s_i$ which can either points up ($s_i = +1$) or down ($s_i = -1$). In its simplest form, it is assumed that only nearest-neighbour spins interact and there is no external field, thus we can write the energy of the system as

$$E = -J \sum_{<ij>} s_i s_j \tag{1}$$

where the sum $< ij >$ is over nearest-neighbour pairs of spins and $J$ is the interaction energy between nearest-neighbour spins.

The Ising model of Eq. 1 has been solved analytically by Lars Onsager [1], but generally analytic techniques are intractable. Therefore it seems natural to proceed by using a computer simulation. A naive implementation would be to calculate the energy of each possible microstate of the system using Eq. (1), and then using the Boltzmann distribution

$$P(microstate) = exp(\frac{-E(microstate)}{k_B T}) \tag{2}$$

we can calculate the macroscopic properties of interest by doing probability summing over all the microstates. However, for a lattice of size $N \times N$ there are $2^{N \times N}$ possible microstates, it is clear that as N increases the computing effort would also increase exponentially if the properties of the system are calculated in this manner. Thus a better numerical alternative would be for our simulation to generate a 'representative' set of microstates with this probability distribution, and then the desired macroscopic quantities can be derived by doing a simple average over this set. The Metropolis-Hastings algorithm [2] is a perfect candidate for doing this.

In this project, we will set $J = 1.0$ which makes the system to be ferromagnetic. In addition, $J/k_B$ will be taken to be unity, and temperature $T$ will be measured in units of $J/k_B$.

## III. Metropolis-Hastings Algorithm

The Metropolis-Hastings Algorithm is best summarised in a flowchart as shown in figure 2.

From the flowchart, we can describe the algorithm step-by-step as follows

1) Create a random system of spins.
2) A spin in the system is selected at random using a random number generator.
3) Find the energy $\Delta E$ required to flip this spin using Eq. 1.
    a) If $\Delta E < 0$, flip the spin and monitor the associated change in the observables.
    b) Else if $exp(-\Delta E/(k_B T) > p$, where p is a uniform random number in the range $[0, 1]$, flip the spin and monitor the associated change in the observables.

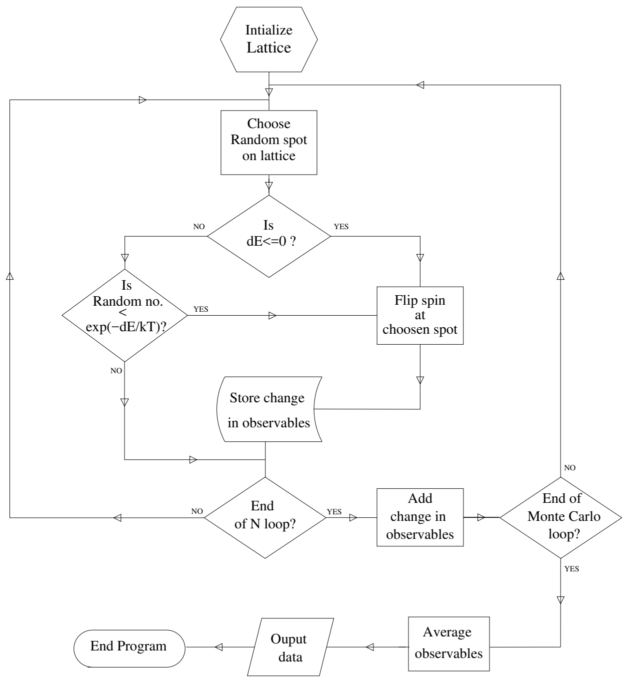
Intialize Lattice
Choose Random spot on lattice
Is dE<=0 ?
NO
YES
Is Random no. < exp(−dE/kT)?
YES
NO
Flip spin at choosen spot
Store change in observables
End of N loop?
NO
YES
Add change in observables
End of Monte Carlo loop?
NO
YES
Average observables
Ouput data
End Program

Fig. 2: Metropolis-Hastings Algorithm flowchart. Figured adapted from [3]

c) Else everything is left unchanged.

4) Repeat the above steps $N \times N$ times. All these steps are contained within a Monte Carlo *sweep*.
5) Repeat for many Monte Carlo sweeps.
6) Record and accumulate the relevant variables, in the 2D Ising model case they are the magnetisation $M$ and energy $E$ of the lattice over time.
7) Output data and run analysis on the data.

The program was implemented so that it can run this procedure for a list of temperature and lattice size values.

After initialization, the process is left to run for $\simeq N^3$ steps before calculation of thermodynamic variables takes place in order for the system to reach thermal equilibrium.

## IV. Calculation of Thermodynamic Variables

The observables of interest, given a temperature $T$ and lattice size $N \times N$, are magnetisation $M$, energy $E$, average magnetisation $\langle M \rangle$, average energy $\langle E \rangle$, average magnetic susceptibility $\langle \chi \rangle$, and average heat capacity $\langle C \rangle$.

The magnetisation and magnetisation per site of the lattice at any given sweep are

$$M(sweep) = \sum_{sites} s_i(sweep) \tag{3}$$

$$M_{persite}(sweep) = \frac{1}{N^2} \sum_{sites} s_i(sweep) \tag{4}$$

The average magnetisation is

$$\langle M \rangle = \frac{1}{no.of sweeps} \sum_{sweeps} M(sweep) \tag{5}$$

The energy and energy per site of the lattice at any given sweep are

$$E(sweep) = -J \sum_{<ij>} s_i(sweep) s_j(sweep) \tag{6}$$

$$E_{persite}(sweep) = \frac{-J}{N^2} \sum_{<ij>} s_i(sweep) s_j(sweep) \tag{7}$$

The average energy is

$$\langle E \rangle = \frac{1}{no.of sweeps} \sum_{sweeps} E(sweep) \tag{8}$$

The magnetic susceptibility is given by

$$\langle \chi \rangle = \frac{1}{T}[\langle M^2 \rangle - \langle M \rangle^2] \tag{9}$$

where $M^2$ is obtained by squaring the magnetisation calculated at each sweep.

The magnetic susceptibility is given by

$$\langle \chi \rangle = \frac{1}{T}[\langle M^2 \rangle - \langle M \rangle^2] \tag{10}$$

where $M^2$ is obtained by squaring the magnetisation calculated at each sweep.

The heat capacity is given by

$$\langle C \rangle = \frac{1}{T^2}[\langle E^2 \rangle - \langle E \rangle^2] \tag{11}$$

where $E^2$ is obtained by squaring the magnetisation calculated at each sweep.

## V. Results

### A. Time evolution

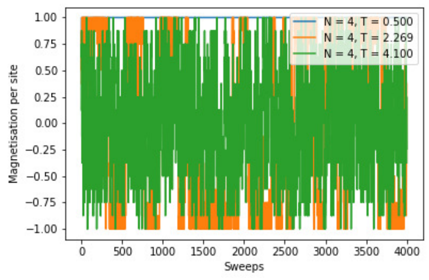

Fig. 3: Time evolution of magnetisation. $N = 4$, $sweeps = 4000$

As we can see from the figures 3, 4, 5, 6, the magnetisation indeed acts as an order parameter. At low temperature, the system will reach a finite value of magnetisation, whereas at hight temperature, it keeps fluctuate about zero. At $T = 2.269$, the magnetisation of the lattice is in an indeterminate state, switching back and forth between $+1$ and $-1$.

### B. Magnetisation

Figure 7 shows beautifully that the shape of the gradient becomes more distinct as the lattice size increases. The behaviour of the magnetisation at low and high temperature limits are as the theory predicted.

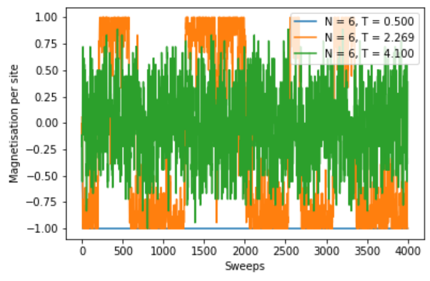

Fig. 4: Time evolution of magnetisation. $N = 6$, $sweeps = 4000$

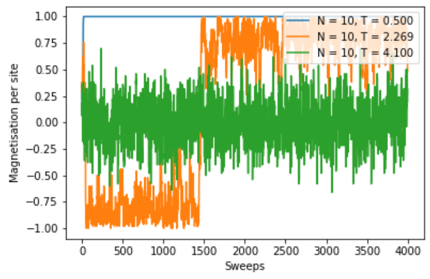

Fig. 5: Time evolution of magnetisation. $N = 10$, $sweeps = 4000$

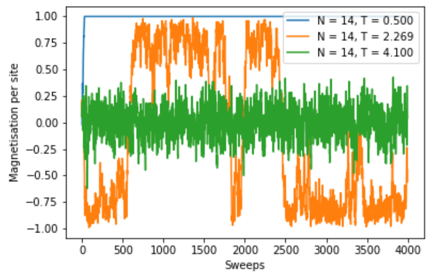

Fig. 6: Time evolution of magnetisation. $N = 14$, $sweeps = 4000$

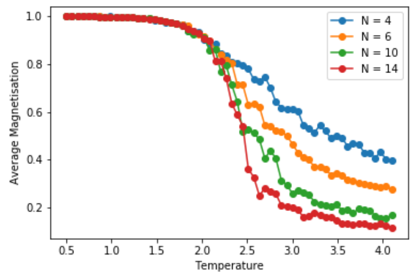

Fig. 7: Effect of temperature on magnetisation for various lattice sizes

## C. Energy

In figure 8 the energy (per site) as a function of temperature can be seen. The gradient of the curve becomes more distinct

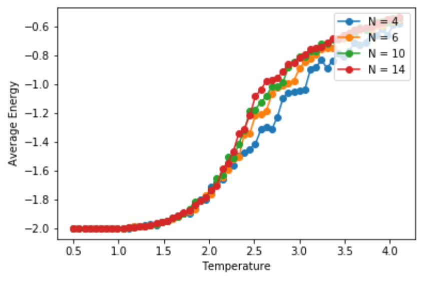

Fig. 8: Effect of temperature on energy for various lattice sizes

as the lattice size increases, but the contrast is not as marked as in the previous magnetisation plot. The divergence of the curves at $T \simeq 2.269$ indicates a possible phase transition.

## D. Susceptibility

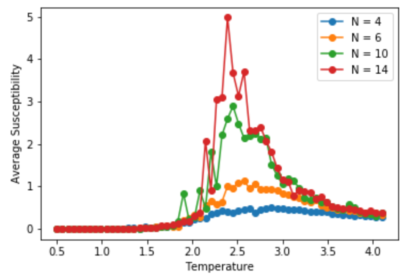

Fig. 9: Effect of temperature on susceptibility for various lattice sizes

A sharp peak is seen in figure 9 at $T = 2.269$. It can be seen that below and above a critical temperature, the magnetic susceptibility is approximately zero, while around the critical temperature it goes to infinity. The plot illustrates clearly that the peak gets sharper with increasing lattice size.

## E. Heat Capacity

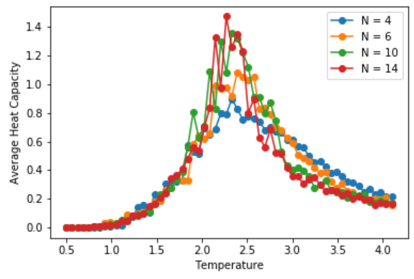

Fig. 10: Effect of temperature on heat capacity for various lattice sizes

The heat capacity tells us how much energy changes with increasing temperature. As seen previously, the energy increases rapidly around $T = 2.269$ so we expect to see a divergence of the specific heat at this transition point. Figure 10 confirms our expectation.

### *F. Finite-size Scaling*

In the infinite lattice size limit, we know that all thermodynamic variables should behave asymptotically as $a|T - T_c|^{\beta}$, with $\beta$ being the critical exponent for each observable.

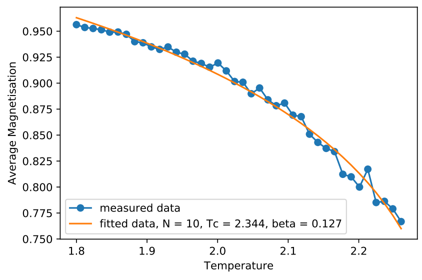

Fig. 11: Fitted and Measured Data

By fitting the data around the critical point, we obtained $\beta = 0.127$ and $T_c = 2.344$ (fig 11). The theoretical values are $\beta = 0.125$ and $T_c = 2.269$.

## REFERENCES

[1] Onsager L., 1944. ”Crystal statistics. I. A two-dimensional model with an order-disorder transition”. *Phys. Rev.*, 65, 117.

[2] Hastings W. K., 1970. ”Monte Carlo Sampling Methods Using Markov Chains and Their Applications”. *Biometrika*, 57, 97.

[3] Kotze J., ”Introduction to Monte Carlo methods for an Ising Model of a Ferromagnet”. arXiv: 0803.0217.

## VI. APPENDIX - PYTHON SOURCE CODE

```python

# coding: utf-8

# In[ ]:


# Import modules
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
import time


# In[ ]:


# Class representing a 2D Ising model
class Ising:
    def __init__(self, N, J, H):
        self.lattice = np.random.choice([-1,1], size=(N,N)) # initial state
        self.N = N # dimension
        self.J = np.abs(J) # interaction energy, non-negative by definition
        self.H = H # external field


# In[ ]:


# Function to calculate energy change when spin(i,j) is flipped, assuming periodic boundary conditions
def energy_change_of_flip(state, pos_i, pos_j):
    return 2*state.lattice[pos_i,pos_j]*(state.J*(state.lattice[pos_i,(pos_j+1)%state.N] + state.lattice[(pos_i-1)%state.N,pos_j] + state.lattice[(pos_i+1)%state.N,pos_j] + state.lattice[pos_i,(pos_j-1)%state.N])+state.H)


# In[ ]:


# Monte Carlo sweep using Metropolis algorithm
def monte_carlo_sweep(state, temperature):
    for i in range(len(state.lattice)**2):
        # pick a spin randomly and calculate energy change if it's flipped
        pos_i = np.random.randint(0,len(state.lattice))
        pos_j = np.random.randint(0,len(state.lattice))
        delta_E = energy_change_of_flip(state,pos_i,pos_j)

        # to flip or not to flip?
        if delta_E < 0 or np.exp(-delta_E/temperature) > np.random.random():
            state.lattice[pos_i,pos_j] *= -1


# In[ ]:


# Function to calculate magnetisation
def calculate_magnetisation(state):
    return np.sum(state.lattice)


# In[ ]:


# Function to calculate energy
def calculate_energy(state):
    interaction_energy = 0.0
    for pos_i in range(len(state.lattice)):
        for pos_j in range(len(state.lattice)):
            interaction_energy += -state.J*state.lattice[pos_i,pos_j]*(state.lattice[pos_i,(pos_j+1)%state.N] + state.lattice[(pos_i-1)%state.N,pos_j] + state.lattice[(pos_i+1)%state.N,pos_j] + state.lattice[pos_i,(pos_j-1)%state.N])
    interaction_energy *= 0.5 # avoid double counting

    magnetisation_energy = -state.H*calculate_magnetisation(state)

    return interaction_energy+magnetisation_energy


# In[ ]:


# Function to normalize data by number of lattice sites
def normalize_data(data, N):
    return data/(N**2)


# In[ ]:


# Metropolis algorithm
def Metropolis_algorithm(N_range, J, H, T_range, no_of_sweeps):
    # arrays to store magnetisation and energy time series
    M = np.zeros((len(N_range),len(T_range),no_of_sweeps))
    E = np.zeros((len(N_range),len(T_range),no_of_sweeps))
    M_normalized = np.zeros((len(N_range),len(T_range),no_of_sweeps))
    E_normalized = np.zeros((len(N_range),len(T_range),no_of_sweeps))
    run_time = np.zeros((len(N_range),len(T_range)))

    # Monte Carlo sweeps
    for N_index in range(len(N_range)):
        for T_index in range(len(T_range)):
            state = Ising(N_range[N_index],J,H)

            t0 = time.time()
            for sweep in range(no_of_sweeps): # run!
                monte_carlo_sweep(state,T_range[T_index])

                M[N_index,T_index,sweep] = calculate_magnetisation(state)
                E[N_index,T_index,sweep] = calculate_energy(state)
                M_normalized[N_index,T_index,sweep] = normalize_data(M[N_index,T_index,sweep],state.N)
                E_normalized[N_index,T_index,sweep] = normalize_data(E[N_index,T_index,sweep],state.N)

                if sweep%1000 == 0: # output checkpoints
                    print('N = {0}, T = {1}, sweep = {2}'.format(N_range[N_index],T_range[T_index],sweep))
            t1 = time.time()
            run_time[N_index,T_index] = t1-t0

    return M, E, M_normalized, E_normalized, run_time


# In[ ]:


# Function to plot evolution of data over time
def plot_time_series(data, N_range, N_index_range, T_range, T_index_range, start_point, stop_point, xname, yname):
    for N_index in N_index_range:
        for T_index in T_index_range:
            plt.plot(range(start_point,stop_point),data[N_index,T_index,start_point:stop_point],label='N = {0}, T = {1:.3f}'.format(N_range[N_index],T_range[T_index]))
        plt.legend(loc='upper right')
        plt.xlabel(xname)
        plt.ylabel(yname)
        plt.savefig('plots/{0} vs {1} N = {2} from {3} to {4}.pdf'.format(yname,xname,N_range[N_index],start_point,stop_point))
        plt.figure()


# In[ ]:


# Function to calculate the average of a thermodynamic variable
def calculate_thermodynamic_variable(data, N_range, T_range, no_of_equilibrating_sweeps):
    var = np.zeros((len(N_range),len(T_range)))

    for N_index in range(len(N_range)):
        for T_index in range(len(T_range)):
            var[N_index,T_index] = np.sum(data[N_index,T_index,no_of_equilibrating_sweeps:])/len(data[N_index,T_index,no_of_equilibrating_sweeps:])
        var[N_index,:] = normalize_data(var[N_index,:],N_range[N_index])

    return var


# In[ ]:


# Function to calculate the average of a derivative thermodynamic variable
def calculate_derivative_thermodynamic_average(data, N_range, T_range, power_of_T, no_of_equilibrating_sweeps):
    var = np.zeros((len(N_range),len(T_range)))

    for N_index in range(len(N_range)):
        for T_index in range(len(T_range)):
            ave = np.sum(data[N_index,T_index,no_of_equilibrating_sweeps:])/len(data[N_index,T_index,no_of_equilibrating_sweeps:])
            squared_ave = np.sum(data[N_index,T_index,no_of_equilibrating_sweeps:]**2)/len(data[N_index,T_index,no_of_equilibrating_sweeps:])
            var[N_index,T_index] = (squared_ave-ave**2)/(T_range[T_index]**power_of_T)
        var[N_index,:] = normalize_data(var[N_index,:],N_range[N_index])

    return var


# In[ ]:


# Function to plot thermodynamic variable against temperature
def plot_temperature_dependence(data, N_range, N_index_range, T_range, T_index_range, xname, yname):
    for N_index in N_index_range:

        plt.plot(T_range[T_index_range[0]:(T_index_range[-1]+1)],data[N_index,T_index_range[0]:(T_index_range[-1]+1)],'-o',label='N = {0}'.format(N_range[N_index]))
        # +1 to include last element
    plt.legend(loc='best')
    plt.xlabel(xname)
    plt.ylabel(yname)
    plt.savefig('plots/{0} vs {1} from {2:.3f} to {3:.3f}.pdf'.format(yname,xname,T_range[T_index_range[0]],T_range[T_index_range[-1]]))
    plt.figure()


# In[ ]:


# Function to calculate the autocorrelation of a variable
def calculate_autocovariation(data, N_range, T_range, no_of_sweeps, no_of_equilibrating_sweeps):
    no_of_sampling_sweeps = no_of_sweeps-no_of_equilibrating_sweeps
    autocovariation = np.zeros((len(N_range),len(T_range),no_of_sampling_sweeps))

    for N_index in range(len(N_range)):
        for T_index in range(len(T_range)):
            ave = np.sum(data[N_index,T_index,:])/no_of_sampling_sweeps
            for tau in range(no_of_sampling_sweeps):
                autocorr = 0.0
                for i in range(no_of_equilibrating_sweeps,no_of_sweeps-tau):
                    autocorr += (data[N_index,T_index,i]-ave)*(data[N_index,T_index,i+tau]-ave)
                autocovariation[N_index,T_index,tau] = autocorr
            if autocovariation[N_index,T_index,0] == 0:
                autocovariation[N_index,T_index,:] = np.ones(no_of_sampling_sweeps)
            else:
                autocovariation[N_index,T_index,:] = autocovariation[N_index,T_index,:]/autocovariation[N_index,T_index,0]

    return autocovariation


# In[ ]:


# Function of magnetisation near Tc
def shape_function(x, a, Tc, b):
        return a*(((Tc-x)/Tc)**b)


# In[ ]:


# Function to fit data
def data_fitting(data, N_range, N_index_range, T_range, T_index_range, guess):
    x_data = np.zeros((len(N_index_range),len(T_index_range)))
    y_data = np.zeros((len(N_index_range),len(T_index_range))) # data sliced by N_index_range and T_index_range
    for N_index in range(len(N_index_range)):
        for T_index in range(len(T_index_range)):
            x_data[N_index,T_index] = T_range[T_index_range[T_index]]
            y_data[N_index,T_index] = data[N_index_range[N_index],T_index_range[T_index]]

    params = np.zeros((len(N_index_range),3))
    errs = np.zeros((len(N_index_range),3))

    for N_index in range(len(N_index_range)):
        popt, pcov = optimize.curve_fit(shape_function,x_data[N_index,:],y_data[N_index,:],guess)
        params[N_index,:] = popt
        errs[N_index,:] = pcov.diagonal()

    return params, errs


# In[ ]:


# Set parameters and run the experiment!!!
Ns = [10]
J = 1.0
H = 0.0
Ts = np.linspace(1.8,2.26,40)
sweeps = 12000
equilibrating_sweeps = 2000

magnetisation, energy, magnetisation_per_site, energy_per_site, run_time = Metropolis_algorithm(Ns,J,H,Ts,sweeps)


# In[ ]:


# Plot time evolution
N_indices = range(len(Ns))
T_indices = [0,19,39]
startpt = 0
stoppt = sweeps
plot_time_series(magnetisation_per_site,Ns,N_indices,Ts,T_indices,startpt,stoppt,'Sweeps','Magnetisation per site')


# In[ ]:


# Plot run time vs N to investigate program complexity
run_time_data = np.zeros(len(Ns))
for N_index in N_indices:
    run_time_data[N_index] = np.average(run_time[N_index,:])
plt.plot(Ns,run_time_data,'-o')
plt.xlabel('Lattice size N')
plt.ylabel('Run time (s)')
plt.savefig('plots/Run time vs Lattice size.pdf')


# In[ ]:


# How total magnetisation fluctuates with time when system is in equilibrium
magnetisation_autocovariance = calculate_autocovariation(magnetisation,Ns,Ts,sweeps,equilibrating_sweeps)


# In[ ]:


# Plot autocovariance of total magnetisation
N_indices = range(len(Ns))
T_indices = range(26,31)
startpt = 0
stoppt = sweeps
plot_time_series(magnetisation_autocovariance,Ns,N_indices,Ts,T_indices,startpt,stoppt,'tau','Autocorrelation')


# In[ ]:


# Calculate thermodynamic variables
average_magnetisation = calculate_thermodynamic_variable(np.abs(magnetisation),Ns,Ts,equilibrating_sweeps)
average_energy = calculate_thermodynamic_variable(energy,Ns,Ts,equilibrating_sweeps)
average_susceptibility = calculate_derivative_thermodynamic_average(np.abs(magnetisation),Ns,Ts,1,equilibrating_sweeps)
average_heat_capacity = calculate_derivative_thermodynamic_average(energy,Ns,Ts,2,equilibrating_sweeps)


# In[ ]:


# Plot thermodynamic variables
N_indices = range(len(Ns))
T_indices = range(len(Ts))
plot_temperature_dependence(average_magnetisation,Ns,N_indices,Ts,T_indices,'Temperature','Average Magnetisation')
plot_temperature_dependence(average_energy,Ns,N_indices,Ts,T_indices,'Temperature','Average Energy')
plot_temperature_dependence(average_susceptibility,Ns,N_indices,Ts,T_indices,'Temperature','Average Susceptibility')
plot_temperature_dependence(average_heat_capacity,Ns,N_indices,Ts,T_indices,'Temperature','Average Heat Capacity')


# In[ ]:


# Fit magnetisation
N_indices = range(len(Ns))
T_indices = range(len(Ts))
parameters, errors = data_fitting(average_magnetisation,Ns,N_indices,Ts,T_indices,[0.,2.269,0.125])
print(parameters, errors)


# In[ ]:


# Plot fitted vs measured data
for N_index in N_indices:
    plt.plot(Ts[:],average_magnetisation[N_index,:],'-o',label='measured data')

    plt.plot(Ts[T_indices],shape_function(Ts[T_indices],parameters[N_index,0],parameters[N_index,1],parameters[N_index,2]),label='fitted data, N = {0}, Tc = {1:.3f}, beta = {2:.3f}'.format(Ns[N_index],parameters[N_index,1],parameters[N_index,2]))
    plt.legend(loc='best')
    plt.xlabel('Temperature')
    plt.ylabel('Average Magnetisation')
    plt.savefig('plots/Fitted vs Measured Magnetisation N = {0}.pdf'.format(Ns[N_index]))
    plt.figure()
```