ND Ising Model

Christopher Soo

March 2025

Contents

1 Abstract

The Ising model is a mathematical model of ferromagnetism in statistical mechanics. In this project, we extend the Ising model to N-dimensions and implement it using C++. The simulation is based on the Metropolis-Hastings algorithm to determine spin flips based on energy changes. The system is initialized with a random configuration and evolves over multiple Monte Carlo sweeps. We analyze the behavior of the system by tracking energy and magnetization across various parameter settings, including different dimensions, temperature ranges (represented by β), and number of sweeps. The results are visualized through Python-based data analysis. This documentation provides an overview of the implementation, functionality, and compilation instructions for the ND Ising Model simulation.

2 Project Documentation

2.1 ND_ising_system.h

This section will explain the functionalities of the classes and functions in the file `ND_ising_system.h`. This is a header file and it includes the declarations for the classes and functions which are defined in `ND_ising_system.cpp`. Note that a lot of the variables have the `long long` data type. This is because we want to allow for high dimensions and large amount of particles.

Libraries:

- `iostream`: Used for standard functions and printing.
- `fstream`: Used for handling and writing files.
- `random`: Used to generate random numbers for the simulation.
- `vector`: Used as the primary data structure for the model.

ND_System Class Variables (Private):

- `state`: A flattened 1D vector used to store spins with length `LEN`. Stores values 0 or 1. We can get the exact spin by using $spin = 2 \cdot \texttt{state}[i] - 1$
- `LEN`: The length of the vector `state`. Calculated as N^{DIM}.
- `N`: The number of particles in the system.
- `DIM`: The number of dimensions.
- `rd`: A Random Device, used to provide a random seed for the Pseudo Random Number Generator.
- `gen`: A Mersenne Twister Pseudo Random Number Generator.

- `dist`: Specifies `gen` to sample from a $Unif(0, LEN - 1)$ as an integer.
- `prob`: Specifies `gen` to sample from a $Unif(0, 1)$ as a real number.

ND_System Class Method Declarations:

- `ND_System`: Constructor that assigns values to variables and calls the `init` method.
- `flatten`: Method that is used to find the index in the 1D flattened vector given a set of multi-dimensional indices. Takes the array `idx_arr` as an input and outputs the corresponding index as a `long long` number.
- `unflatten`: The inverse of flatten, method that is used to find a set of multi-dimensional indices given the index in the 1D flattened vector. Takes the array `idx_arr` by reference and the number `idx` and updates `idx_arr` with the corresponding multi-dimensional indices.
- `energy_tot`: Method that calculates the total energy of the current state of the vector. It uses the current `state` vector which is a member variable of the class and outputs the total energy as a `long long` number.
- `magnet_tot`: Method that calculates the total magnetisation of the current state of the vector. It uses the current `state` vector which is a member variable of the class and outputs the total magnetisation as a `long long` number.
- `init`: Method that initialises the system. It uses the current `state` vector which is a member variable of the class and populates the empty `state` with 0's or 1's.
- `MC_SWEEP`: Method that uses the Metropolis-Hastings algorithm to accept/reject spins based on the change in energy. It uses the current `state` vector which is a member variable of the class and beta that affects the accept/reject probability and updates `state` accordingly.

2.2 ND_ising_system.cpp

This section will explain the implementations of the declarations in `ND_ising_system.h` and also includes libraries from the same file.

`ND_System` Class Method Implementations:

- `ND_System`: Assigns:
  - DIM→DIM
  - N→N
  - LEN→LEN
  - state→state(LEN)
  - gen→gen(rd())
  - dist→dist(0, LEN-1)
  - prob→prob(0.0, 1.0)

  and runs `init`
- `flatten`: Uses the formula: $i_{\texttt{state}} = \sum_{z=0}^{DIM} i_z \cdot N^{DIM-z}$ where $i_0 \dots i_{DIM}$ are multi-dimensional indices.
- `unflatten`: Uses the formula: $i_z = \frac{i_{\texttt{state}}}{N^z} \bmod N$.
- `energy_tot`: Uses the formula $E = -\sum_{\langle i,j \rangle} S_i S_j$
- `magnet_tot`: Uses the formula $M = \sum_i S_i$
- `init`: Gives every entry a 50% chance of either being 1 or 0 i.e `state`$[i] = Bern(\frac{1}{2})$ for all $i \in N_{[0,LEN-1]}$.
- `MC_SWEEP`: Uses the Metropolis-hastings Algorithm that follows these instructions:
  1. Pick a random index i using `dist`.
  2. Unflatten it into set of multi-dimensional indices $i_0 \dots i_z$.
  3. Calculate $\Delta E = 2S_i \sum_{\langle j \rangle} S_j$ where S_j are the direct neighbors of S_i. This could be done by finding all combinations of $i_j \pm 1$ for all j.
  4. Calculate $P = e^{-\beta \Delta E}$ and generate a random number y using `prob`. If $y < P$ then the algorithm accepts flips `state`[i].

2.3 ND_sweep.cpp

This section will explain the functionalities of the functions in `ND_sweep.cpp`. This file includes from `ND_system.h` and serves as the main code to test simulations for individual sweeps.

- `pow_int`: This function returns the power of a number. It takes two integers as inputs; the base and the power and returns a `long long` number.
- `main`: This is the entry point of the file when the executable is run after compiling. The purpose of this function is to track the energy and magnetisation after each individual `MC_SWEEP` and dump it into a csv file to be analysed in Python. This function accepts 4 command-line arguments:
  1. N: the number of particles (integer)
  2. DIM: the number of dimensions (integer)
  3. BETA: the parameter used in the accept / reject probability (float)
  4. SWEEP: the number of iterations of `MC_SWEEP` (integer)

2.4 ND_ising_model.cpp

This section will explain the functionalities of the functions in `ND_ising_model.cpp`. This file includes from `ND_system.h` and serves as the main code to test simulations for individual configurations.

- `pow_int`: This function returns the power of a number. It takes two integers as inputs; the base and the power and returns a `long long` number.
- `main`: This is the entry point of the file when the executable is run after compiling. The purpose of this function is to track the energy and magnetisation after each configuration, which is tracked after a certain number of iterations of `MC_SWEEP` and dump it into a csv file to be analysed in Python. This function accepts 5 command-line arguments:
  1. N: the number of particles (integer)
  2. DIM: the number of dimensions (integer)
  3. SWEEP: the number of iterations of `MC_SWEEP` (integer)
  4. CONFS: the number of configurations (integer)
  5. BETA: the parameter used in the accept / reject probability (float)

2.5 Compilation Instructions

To compile you can directly run: `g++ -o main ND_ising_model.cpp ND_ising_system.cpp` directly in the main terminal and run: `./main [N] [DIM] [SWEEP] [CONFS] [BETA]` right after where the variables in brackets are the parameters. You can also navigate to the file `view.ipynb` and run the code directly from there using subprocesses (code is provided to plot graphs). The same applies to `ND_sweep.cpp`.

3 1D Ising Model Analysis

3.1 Finding Beta Range

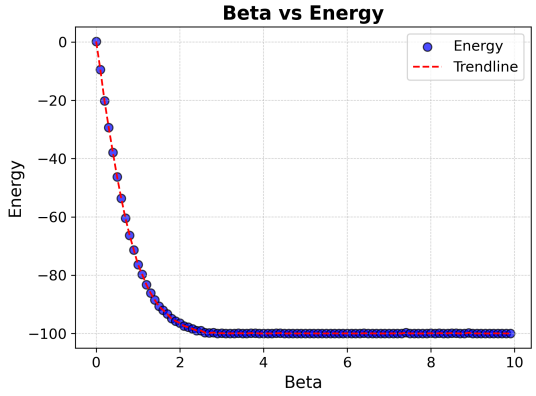

Figure 1: Energy vs Beta

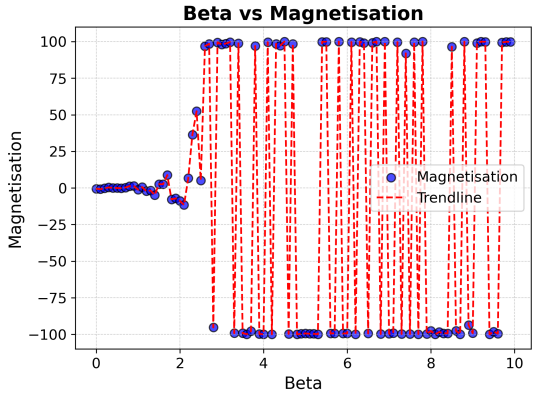

Figure 2: Magnetisation vs Beta

From the plots above we see that in figure 1, the energy starts stabilising at $\beta = 4$ and in figure 2 we also see that the critical temperature is somewhere around $\beta = 2$ too based on the magnetisation. Based on this, we can choose to test cases where $\beta \in [0, 4]$

3.2 Finding Sweeps

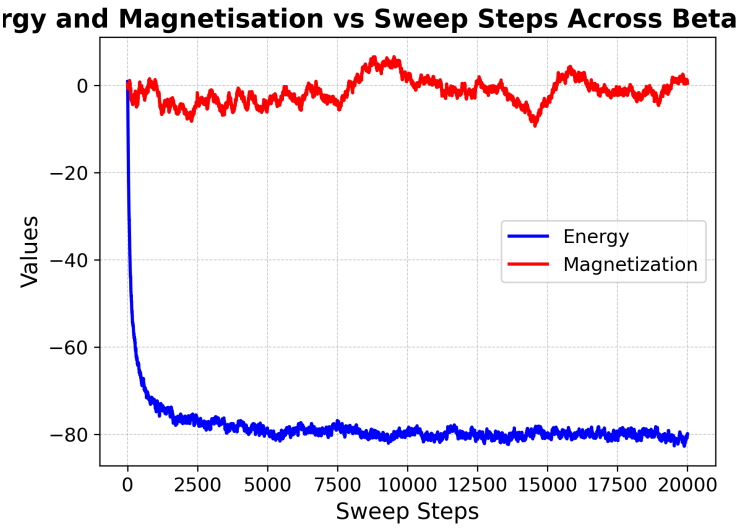

Figure 3: Energy and Magnetisation vs Sweep Steps Across Beta

From the plot above, we can see that the magnetisation fluctuates randomly but the energy starts stabilising at 5000 iterations and fully stabilises at 10000 iterations. Based on this, we can choose 10000 `MC_SWEEP` iterations.

3.3 Finding Configurations

This project uses 1000 configurations, and it's enough to generate a good generalisation. A huge limitation of this project is that the algorithm is not efficient enough to run configurations for more than 1000. One suggestion is to include optimised code for time efficiency for higher amount of configurations.

3.4 Distributions of Energy and Magnetisation

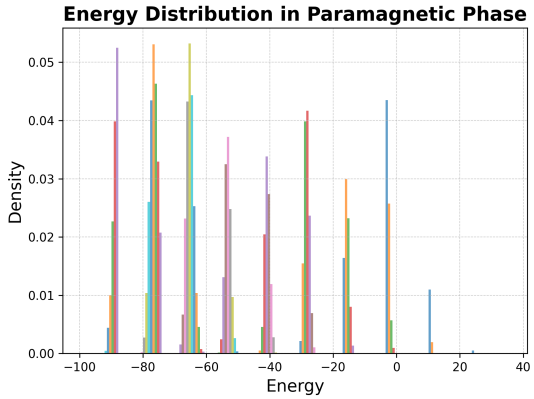

Figure 4: Energy Distribution for Paramagnetic Phase

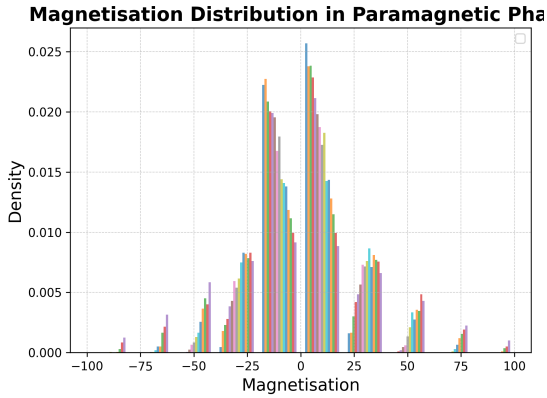

Figure 5: Magnetisation Distribution for Paramagnetic Phase

From the plots above we see that the distribution of the energy at the paramagnetic phase ($\beta \in [0, 1.5]$) is pretty much random and has a lot of noise. While the magnetisation is normally distributed around 0.

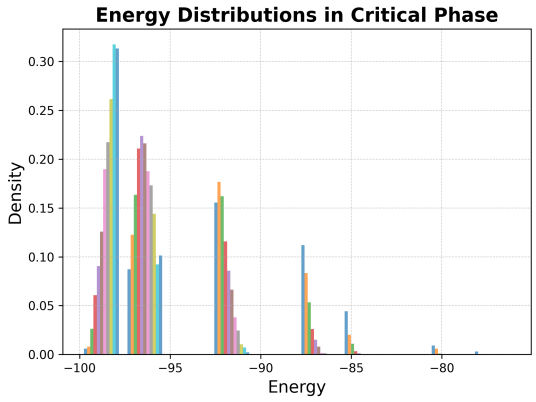

Figure 6: Energy Distribution for Critical Phase

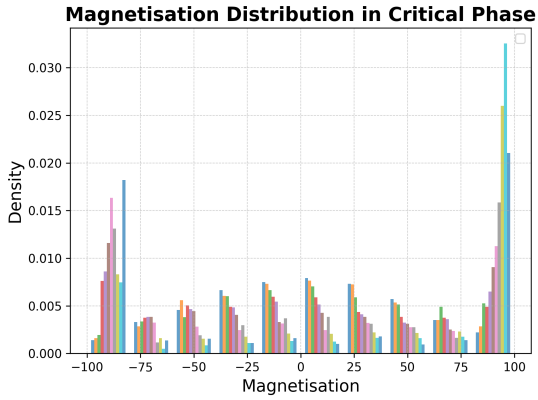

Figure 7: Magnetisation Distribution for Critical Phase

From the plots above we see that the distribution of the energy at the critical phase ($\beta \in [1.5, 2.5]$) is slowly starting to be skewed to the negative values, and the magnetisation is slowly starting to show bimodal signs.

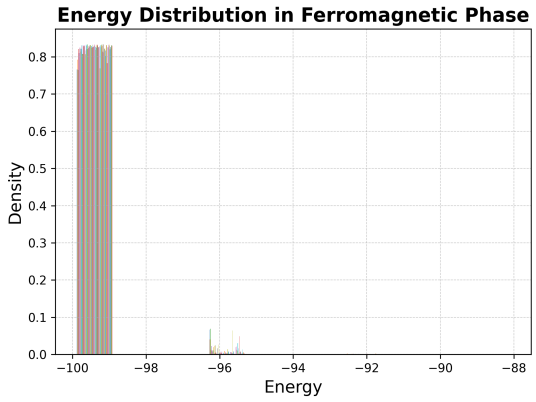

Figure 8: Energy Distribution for Ferromagnetic Phase

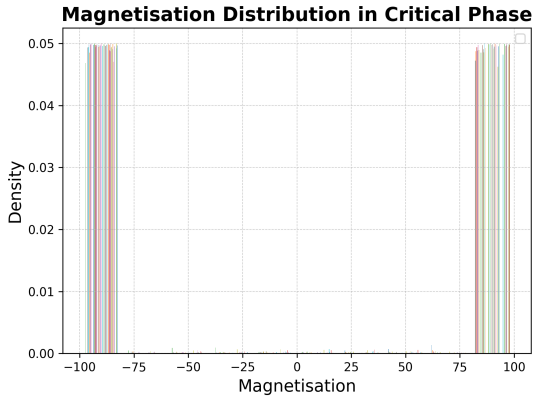

Figure 9: Magnetisation Distribution for Ferromagnetic Phase

From the plots above we see that the distribution of the energy at the ferromagnetic phase ($\beta \in [2.5, 4]$) is heavily skewed to negative values. While the magnetisation is essentially bimodal.

4 2D Ising Model Analysis

4.1 Finding Beta Range

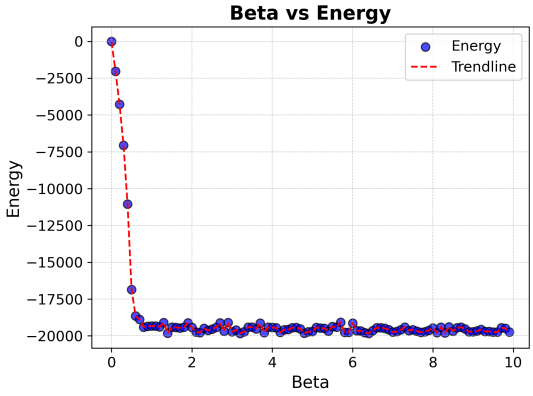

Figure 10: Energy vs Beta

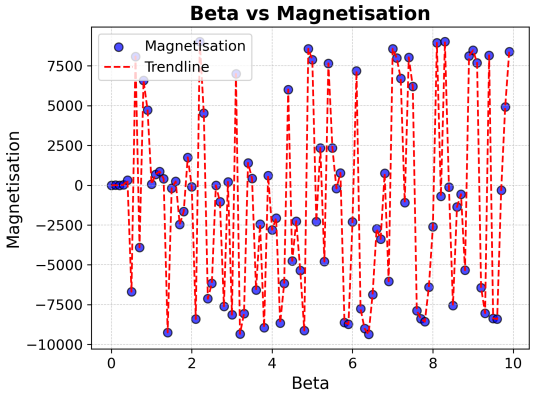

Figure 11: Magnetisation vs Beta

From the plots above we see that in figure 1, the energy starts stabilising at $\beta = 1$ and in figure 2 we also see that the critical temperature is not easy to see because there are a lot of noise and fluctuations. We can use the previous range as a reference, which is $\beta \in [0, 4]$

4.2 Finding Sweeps

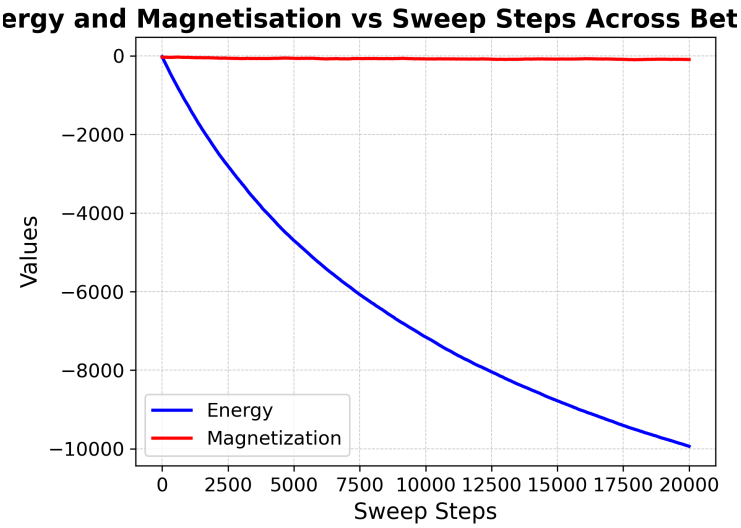

Figure 12: Energy and Magnetisation vs Sweep Steps Across Beta

From the plot above, we can see that the magnetisation is very stable and the energy decreases logarithmically and it starts stabilising at 20000 iterations. Based on this, we can choose 20000 `MC_SWEEP` iterations.

4.3 Distributions of Energy and Magnetisation

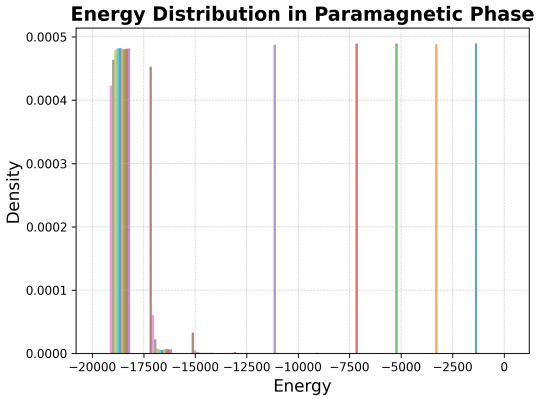

Figure 13: Energy Distribution for Paramagnetic Phase

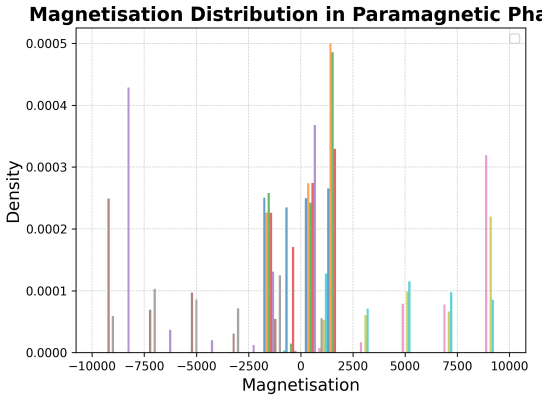

Figure 14: Magnetisation Distribution for Paramagnetic Phase

From the plots above we see that the distribution of the energy at the paramagnetic phase ($\beta \in [0, 1.5]$) is pretty much random with a big cluster in the negative values. While the magnetisation is normally distributed around 0 with more noise and variance in comparison to 1D.

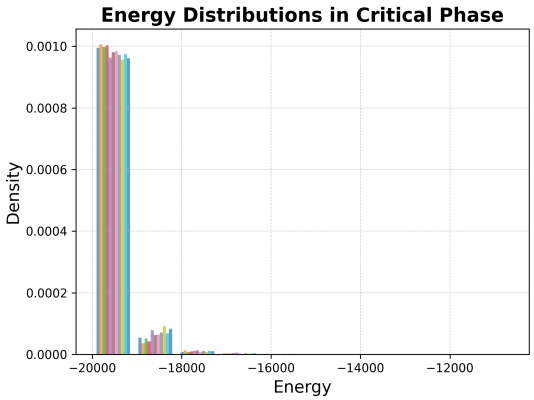

Figure 15: Energy Distribution for Critical Phase

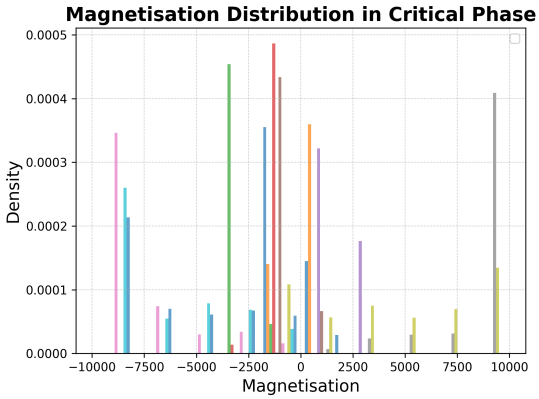

Figure 16: Magnetisation Distribution for Critical Phase

From the plots above we see that the distribution of the energy at the critical phase ($\beta \in [1.5, 2.5]$) heavily skewed to the negative values, and the magnetisation is still normal with a higher variance and it has some bimodal features as well.

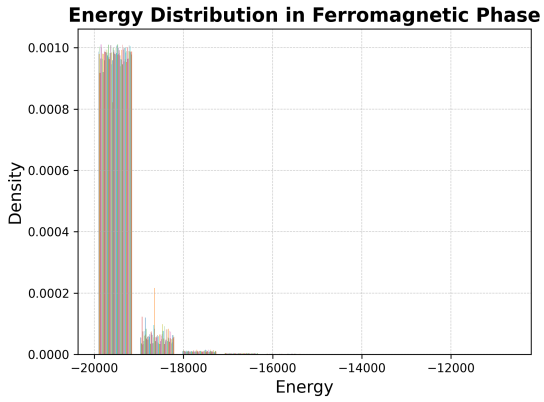

Figure 17: Energy Distribution for Ferromagnetic Phase

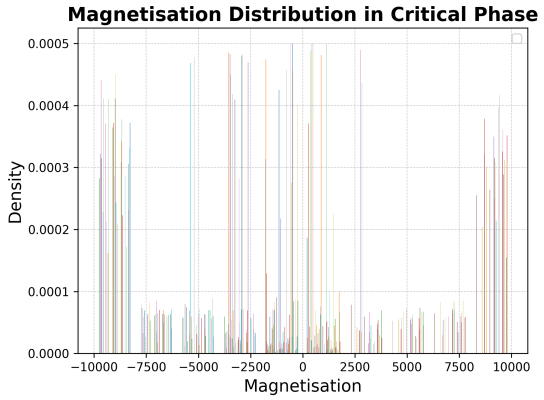

Figure 18: Magnetisation Distribution for Ferromagnetic Phase

From the plots above we see that the distribution of the energy at the ferromagnetic phase ($\beta \in [2.5, 4]$) is heavily skewed to negative values. While the magnetisation is forming more into bimodal with some values in the middle still present.

5 N-Dimensional Ising Model

The code provided by this project allows for higher dimensions but due to limitations in computational power, analysis in 3D and beyond will not be covered. Future studies can use the code as a refernce and extend the investigation to higher dimensions.

6 Conclusion

From the two investigations; 1D and 2D. We conclude that:

- The energy is inversely proportional to β and in higher dimensions, it has a sharper curve.
- The magnetisation has a critical point before becoming bimodal. Before the critical point, it fluctuates around 0. After, it fluctuates in the extremes. In higher dimensions, it takes more iterations / configurations to converge, and it has more noise and variance too.
- The number of sweeps needed to converge increases as the dimensions increase. So does the amount of configurations needed.