One- and Two-Dimensional Ising Model

L.E.N. Baakman*
l.e.n.baakman@student.rug.nl

September 16, 2016

1. Introduction

A large number of systems change their macroscopic properties at thermal equilibria. For example magnetic atoms align them selves to form a magnetic material at low temperature or high pressure. When modeled mathematically, these phase transitions only occur in infinitely large systems [3]. This paper investigates a simulation of a finite system that models a system that exhibits macroscopic properties at its thermal equilibrium, the Ising ferromagnet to be exact.

Section 1.1 introduces the Ising model of ferromagnetism, the next section discusses the Metropolis Monte Carlo method that is used to solve the Ising model numerically.

1.1. Ising Model

A magnet can be modeled as a large collection of electronic spins. In the Ising model spins point either up, $\mathrm{s} = +1$, or down, $\mathrm{s} = -1$ [8]. At high temperatures the spins point in random directions, consequently the magnetization is approximately zero. At a low enough temperature all spins in the two-dimensional model align themselves, this effect is called spontaneous magnetization. The temperature at which this phase transition occurs is called the critical temperature, T_c [1]

Section 1.1.1 and 1.1.2 introduce the one- and two-dimensional Ising model, respectively.

1.1.1. One-Dimensional Model

Ising [2] introduced a model consisting of a one-dimensional lattice of spin variables. Contrary to the two dimensional model this model does not exhibit state transitions. The Hamiltonian of the one dimensional Ising model with the set spins $\mathcal{S} = \{\mathrm{s}_1, \ldots, \mathrm{s}_\mathrm{N}\}$ is

$$\mathcal{H}(\mathcal{S}) = -\mathcal{J} \sum_{\langle i,j \rangle} \mathrm{s}_i \mathrm{s}_j - \mathrm{h} \sum_i \mathrm{s}_i. \tag{1}$$

Where $\langle i, j \rangle$ is a nearest neighbor pair. The nearest neighbors of s_i in the one dimensional model are s_{i-1} and s_{i+1}. $\mathcal{J}$ specifies the strength of the interactions between the spins. In a ferromagnetic model, $\mathcal{J} > 0$ neighboring spins prefer to be parallel. In the antiferromagnetic model, $\mathcal{J} < 0$ spins prefer a direction different to that of their neighbors. The constant h represents the external magnetic field. Spins prefer to align themselves with the direction of h, for example when $\mathrm{h} > 0$ spins prefer to be positive.

In the following the zero-field ferromagnetic model, i.e. $\mathcal{J} = 1$ and $\mathrm{h} = 0$, is considered. The energy E of a configuration of spins, $\mathcal{S}$, in this model is given by

$$E(\mathcal{S}) = \sum_{n=1}^{\mathrm{N}-1} \mathrm{s}_n \mathrm{s}_{n+1}.$$

The probability of realizing a configuration of spins $\mathcal{S}$ with energy E is defined as

$$P(\mathcal{S}) = \frac{1}{Z} \exp\left[-E(\mathcal{S}) \frac{1}{T}\right], \tag{2}$$

*Master Profile: Computing Science
Student Number: s1869140

where T represents the temperature, $\beta = {}^1/_T$ and Z is the partition function:

$$Z = \sum_{\{\mathrm{s}_1,\ldots,\mathrm{s}_\mathrm{N}\}} \exp\left[-E(\mathcal{S})\beta\right]. \tag{3}$$

Both the one- and two-dimensional Ising model can be solved analytically. Under free end boundary conditions, i.e. the boundary spins, s_1 and s_N, only observe one neighbor [5], the analytical solution of equation (3) is

$$Z = \left(2\cosh\beta\right)^\mathrm{N}. \tag{4}$$

The average energy can be expressed as a function of Z [6]

$$U = \frac{1}{Z} \cdot \sum_{\mathcal{S}_i \in \Phi} E\left(\mathcal{S}_i\right) \cdot \exp\left[-\beta E\left(\mathcal{S}_i\right)\right],$$

where Φ is the set of all possible configurations, i.e. $\Phi = \{\mathcal{S}_1, \ldots, \mathcal{S}_{2^\mathrm{N}}\}$. Observing that

$$\frac{\partial Z}{\partial \beta} = \sum_{\mathcal{S}_i \in \Phi} -E\left(\mathcal{S}_i\right) \exp\left[-\beta E\left(\mathcal{S}_i\right)\right],$$

and by following the steps presented in appendix A.1 it can be found that

$$U = -\frac{\partial \ln\left[Z\right]}{\partial \beta} = -\mathrm{N} \cdot \tanh(\beta).$$

Consequently ${}^U/_\mathrm{N} = -\tanh(\beta)$.

The specific heat describes how the average energy changes as a function of the temperature. Consequently

$$C = \frac{\partial U}{\partial T} = \mathrm{N}\left(\frac{\beta}{\cosh(\beta)}\right)^2$$

as shown in appendix A.2 [9] from this follows

$$\frac{C}{\mathrm{N}} = \left(\frac{\beta}{\cosh(\beta)}\right)^2.$$

1.1.2. Two-Dimensional Model

We consider as the two-dimensional Ising model a square lattice whose lattice sites are occupied by spins. Each spin is either positive or spin [3]. The Hamiltonian of the two-dimensional model is the same as the one of the one-dimensional model presented in equation (1). The pairs of nearest neighbors are now found by looking at the four-connected neighbors, i.e. the nearest neighbors of spin $\mathrm{s}_{i,j}$ are $\mathrm{s}_{i-1,j}$, $\mathrm{s}_{i+1,j}$, $\mathrm{s}_{i,j-1}$ and $\mathrm{s}_{i,j+1}$. The energy of a configuration $\mathcal{S}_n$ that has $\mathrm{N} = \mathrm{d} \times \mathrm{d}$ spins is computed as

$$\begin{aligned} E\left(\mathcal{S}_n\right) = &-\sum_{i=1}^{\mathrm{d}-1}\sum_{j=1}^{\mathrm{d}} \mathrm{s}_{i,j}\mathrm{s}_{i+1,j} \\ &-\sum_{i=1}^{\mathrm{d}}\sum_{j=1}^{\mathrm{d}-1} \mathrm{s}_{i,j}\mathrm{s}_{i,j+1}. \end{aligned} \tag{5}$$

The infinite Ising system, i.e. $\mathrm{N} = \infty$, with $\mathrm{d} \geq 2$ has a phase transition when no external magnetic field is present, i.e. $\mathrm{h} = 0$. In this case the the spins align themselves parallel to each other in the thermal average below the critical temperature T_c. In this way they generate macroscopic magnetization. The magnetization itself can be either positive or negative. The phase transition is critical, which means that as the temperature increases, the magnetization, M, of the system changes continuously towards zero [3].

The two-dimensional Ising model has been solved analytically by Onsager [7]. He showed that the average magnetization per spin on a infinite two-dimensional lattice, i.e. $\mathrm{N} = \infty$, is

$$\frac{\mathrm{M}}{\mathrm{d}^2} = \begin{cases} \left(1 - \sinh^{-4}(2\beta)\right)^2 & \text{if } T < T_c \\ 0 & \text{if } T > T_c \end{cases} \tag{6}$$

where

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

Given equation (3) solving the the Ising model is relatively simple. To find which configurations of spins result in an equilibrium one only needs to try them all. Unfortunately the computational complexity of this operation is exponential in N, the number of spins.

To be exact, a lattice with N spins has 2^{N} possible configurations, computing E according to equation (5) for one configuration takes 2N steps. This leads to $2\mathrm{N}2^{\mathrm{N}}$ computation steps [3]. Solving the problem with the Metropolis Monte Carlo method circumvents the issues caused by the high computational complexity.

1.2. Metropolis Monte Carlo

Monte Carlo methods rely on random sampling to obtain numerical results. They are often used to solve problems that might be deterministic in principle but are difficult to solve with other approaches. One of the applications of Monte Carlo experiments is sampling, i.e. generating draws from some probability distribution [4].

In the context of the Ising model one could naively consider using a few randomly generated states to compute the partition function. However the central limit theorem tells us that these states have an energy that is approximately $\mathcal{O}(\sqrt{\mathrm{N}})$ for sufficiently large N. However the states that we are interested in have an energy of the order $\mathcal{O}(\mathrm{N})$, which means that they are not generated at all by the naive method.

Consequently we need some way to generate the physically relevant states. This can be done by relaxing some configuration into a thermal equilibrium by generating from it a new sequence of states. This requires a transition probability $W\left(\mathcal{S}_i \to \mathcal{S}_j\right)$ from configuration $\mathcal{S}_i$ to configuration $\mathcal{S}_j$. In thermal equilibrium the probability of finding a given configuration is presented in equation (2). As we require $P\left(\mathcal{S}_i\right)$ to be stationary in thermal equilibrium we get the detailed balance:

$$W\left(\mathcal{S}_i \to \mathcal{S}_j\right) \exp\left[-E(\mathcal{S}_i)\beta\right] = W\left(\mathcal{S}_j \to \mathcal{S}_i\right) \exp\left[-E(\mathcal{S}_j)\beta\right]. \quad (7)$$

The function $W(\cdot)$ needs to cover the entire configuration space. The Metropolis algorithm is one of the algorithms that ensures this [3].

This algorithm starts in some initial configuration, it then moves to subsequent configurations by flipping one randomly selected spin with a probability defined by $W(\cdot)$. This is repeated for a given number of steps. Generally one should give the system as number of steps to relax into an interesting state before actually using the generated states.

Section 2 discusses how the Metropolis Monte Carlo method is used to solve the Ising model. In section 3 the run experiments are introduced, their results are presented in section 4. Section 5 discusses the found results and section 6 concludes this paper.

2. Method

Algorithm 1 presents the pseudo code of the Metropolis Monte Carlo algorithm applied to the Ising problem. This section starts by discussing the input of this algorithm and then introduces the functions used in algorithm 1 one by one. It should be noted that the discussed algorithm is agnostic to the dimensionality of the model.

Algorithm 1: MMC($\mathcal{S}_{init}, N_{iterations}$)

input : $\mathcal{S}_{init}$ the initial configuration
$N_{iterations}$ number of iterations

$\mathcal{S}_{cur} := \mathcal{S}_{init}$
for $i = 0$ **to** $N_{iterations}$ **do**
  s := `selectRandomSpin(`$\mathcal{S}_{cur}$`)`
  $\mathcal{S}_{pot}$:= `flipSpin(s,` $\mathcal{S}_{cur}$`)`
  $\mathcal{S}_{cur}$:= `select(` $\mathcal{S}_{cur}$`,` $\mathcal{S}_{pot}$ `)`

Algorithm 1 requires an initial configuration $\mathcal{S}_{init}$ as input, this configuration is a representation of the system in its initial state. If the final configuration of a system with a lower temperature is available when `MMC()` is called, this configuration is used as $\mathcal{S}_{init}$. Otherwise a configuration with only positive spins is used. The parameter $N_{iterations}$ indicates how many configurations are generated. Before the start

of the first loop the the current configuration is set to the initial configuration.

`selectRandomSpin()` selects on spin randomly from the spins in $\mathcal{S}_{cur}$. The potential configuration, $\mathcal{S}_{pot}$ is a copy of $\mathcal{S}_{cur}$ with the selected spin, s, flipped. This new configuration is generated by `flipSin()`. The new current configuration is selected by `select()`. The pseudo code of this function is presented in algorithm 2.

Algorithm 2: select($\mathcal{S}_{cur}, \mathcal{S}_{pot}$)

input : $\mathcal{S}_{cur}$ the current configuration
$\mathcal{S}_{pot}$ the potential configuration
output: $\mathcal{S}_{new}$ the selected configuration

$\Delta E :=$ `computeDeltaE(` $\mathcal{S}_{cur}$`,` $\mathcal{S}_{pot}$ `)`
$\xi := \exp\left[-\beta \Delta E\right]$
$\theta :=$ `randomNumber(0, 1)`
if $\xi > \theta$ **then** $\mathcal{S}_{new} := \mathcal{S}_{pot}$
else $\mathcal{S}_{new} := \mathcal{S}_{cur}$

Given two configurations, $\mathcal{S}_{pot}$ and $\mathcal{S}_{cur}$, `select()` selects with which configuration the simulation should continue. To this end ΔE, the difference in energy between the two configurations, is computed. It is not possible to compute the energy of the two configurations according to equation (5), as that would require all states. However writing equation (1) as

$$\mathcal{H}(\mathcal{S}) = -\mathrm{s}_i \sum_{j \in \mathcal{N}(\mathrm{s}_i)} \mathrm{s}_j + \textit{remainder} \quad (8)$$

where $\mathcal{N}(\mathrm{s}_i)$ is the neighborhood of s_i allows us to see that since only s_i changes the *remainder* of $\mathcal{H}(\mathcal{S}_{cur})$ and $\mathcal{H}(\mathcal{S}_{pot})$ are the same [3]. Thus only the first term in equation (8) is relevant for the computation of ΔE, consequently we find:

$$\Delta E = -2 \cdot \mathrm{s}_i \sum_{j \in \mathcal{N}(\mathrm{s}_i)} \mathrm{s}_j. \quad (9)$$

If a potential configuration becomes the current configuration is determined based on two values: θ and the Boltzman factor ξ. The first is sampled from a pseudo random uniform distribution with the range $(0, 1)$. The second is defined as

$$\xi = \exp\left[-\beta \Delta E\right]. \quad (10)$$

If the transition from $\mathcal{S}_{cur}$ to $\mathcal{S}_{pot}$ decreases the energy of the system the Boltzman factor is greater than one and as $\theta \in (0, 1)$ potential states with an energy that is lower than that of the current state are always accepted. The guard of the `if` also ensures that state transitions that increase the energy, i.e. $\xi < 1$, are not necessarily discarded.

The implementation of the presented algorithm can be found in listings 1 to 3 in appendix C.1.

3. Experiments

This section introduces the experiments we ran with the model introduced in the previous sections. The required random numbers were generated with a Mersenne Twister with seed 0. To give the system time to relax into the interesting states we perform $^1/_{10} \cdot N_{iterations}$ Monte Carlo steps, before actually taking the samples used to compute the results.

The experiments we ran with the one and two dimensional model are discussed in sections 3.1 and 3.2, respectively.

3.1. One-Dimensional Model

In the one-dimensional model we are interested in both the average energy and the specific heat per spin in the following parameter space $T = 0.2, 0.4, \ldots, 4$, $\mathrm{N} = 10, 100, 1000$ and $N_{iterations} = 1000, 10000$.

U, the average energy is given by

$$U = \frac{1}{\#\Omega} \sum_{\mathcal{S}_i \in \Omega} E(\mathcal{S}_i), \quad (11)$$

where $\Omega = \left\{\mathcal{S}_1, \ldots, \mathcal{S}_{N_{samples}}\right\}$ is the set of configurations generated during the Monte Carlo

steps. C, the specific heat is defined as

$$C = \beta^2 \left(\frac{1}{\#\Omega} \left(\sum_{\mathcal{S}_i \in \Omega} E^2(\mathcal{S}_i) \right) - U^2 \right). \quad (12)$$

Furthermore we will compare the results of the simulation with the analytical solution presented in section 1.1. To compare the numerical and analytical results the mean accuracy of the specific heat and the average energy per spin are computed. The accuracy of a variable where ν and α represent the numerically and analytically found values, respectively, is

$$\text{accuracy} = 1 - \left| \frac{|\nu - \alpha|}{\alpha} \right|. \quad (13)$$

The discussed experiment is implemented in listing 8. The implementation of equations (11) to (13) are presented in listings 4 to 6, respectively. All mentioned listings can be found in appendix C.

3.2. Two-Dimensional Model

The two-dimensional model should exhibit a phase transition. Consequently we are not only interested in the average energy and specific heat per spin but also the average magnetization per spin. The magnetization of the Ising model can be computed as:

$$\mathrm{M} = \frac{1}{\#\Omega} \sum_{\mathcal{S}_i \in \Omega} \sum_{\mathrm{s}_j \in \mathcal{S}_i} \mathrm{s}_j. \quad (14)$$

The following parameter space is used: $T = 0.2{,}0.4, \ldots, 4$, $\mathrm{d} = 10{,}50{,}100$ and $N_{iterations} = 1000{,}10000$. The found average average magnetization per spin is compared with the analytical solution presented in equation (6).

The discussed experiment and equation (14) are presented in listings 7 and 9 in appendix C, respectively.

4. Results

This section presents the results of the experiments with the one- and two- dimensional model discussed in section 3, in section 4.1 and section 4.2, respectively.

Table 1: *Average accuracies of the one-dimensional simulation.*

	Number of Spins (N)		
$N_{samples}$	10	100	1000
1000	-0.8	-1	-0.10
1000	0.9	0.8	0.7

4.1. One-Dimensional Model

Tables 2 to 7 in appendix B present the results of the experiment with the one dimensional model. The average accuracies, without the $\pm\infty$ are presented in table 1.

In tables 2 and 3 we observe that the accuracy of the 1D simulation is reasonable for temperatures greater than 0.4. In the simulation with N = 10. The simulation with N = 100 starts being accurate at $T = 1.4$ if $N_{samples} = 1000$ and at $T = 0.8$ if $N_{samples} = 10000$. If we increase the number of spins to N = 1000 we only find reasonable accuracy with $N_{samples} = 10000$ for $T > 1.6$.

For all values of N we find that the accuracy improves we find that the accuracy improves as the number of samples increases.

4.2. Two-Dimensional Model

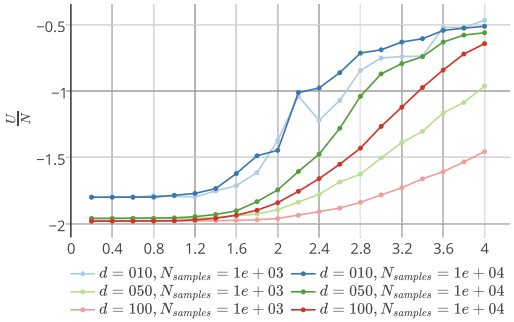
The average energy, specific heat and average magnetization per spin for the different combinations of N and $N_{samples}$ can be found in figure 1.

In figure 1a we observe that the average energy per spin is hardly influenced by the number of samples for $N_{samples} = 10$. As the number of spins in the simulation increases, the difference between the simulation with $N_{samples} = 1 \times 10^3$ and $N_{samples} = 1 \times 10^4$ increases. In general we observe that the average energy per spin increases as the temperature increases. In equation (11) the phase transition that we expect at $T_c \approx 2$ is indicated

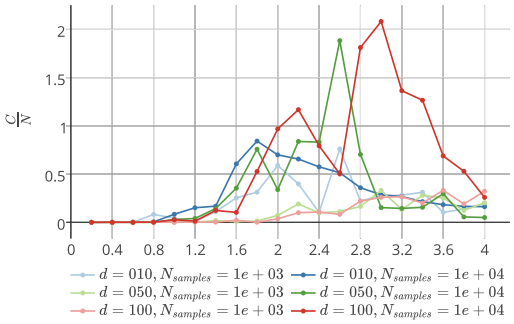
by an increase in the average energy per spin, after $^{U}/_{\mathrm{N}}$ has been constant for $T << T_c$.

In figure 1b we observe a bell-shaped curve in the specific heat per spin around $T = 2$. The curve is more defined when $N_{samples}$ is higher and when the number of spins in the simulation increases. The highest points of these bell curves are found near $T = T_c$, i.e. at the phase transition.

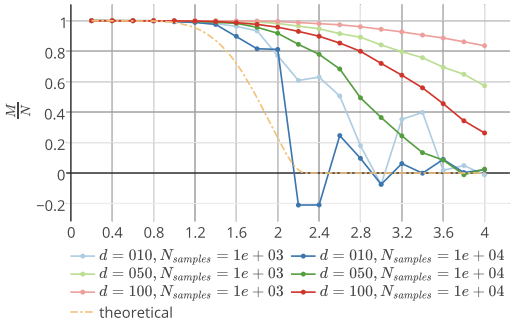
Comparing the measured average magnetization per spin in figure 1c with the theoretical value in the same figure we observe that the curves reflecting the results of the simulation are less steep. Furthermore the smaller simulations seem to give a better approximation than the simulation with a lot of spins. In figure 1c the phase transition is clearly illustrated by the theoretical line. Nearly all lines in this figure illustrate that $^{\mathrm{M}}/_{\mathrm{N}}$ is positve for low temperatures, it then decreases around $T = T_c$ and stays zero for temperatures that are a bit higher than the critical temperature. The last observation only holds for the theoretical infinite system. Most of the simulated systems do not reach $^{\mathrm{M}}/_{\mathrm{N}} = 0$.



(a) *Average energy per spin.*



(b) *Specific heat per spin.*



(c) *Average magnetization per spin.*

Figure 1: *The (a) average energy, (b) specific heat and (c) average magnetization per spin in a 2D Ising model with* $\mathrm{d} = 10, 50, 100$ *and* $N_{samples} = 1000, 10000$.

5. Discussion

In this section the results of the two experiments, presented in section 4, are discussed.

5.1. One-Dimensional Model

We have observed that the accuracy increases as the temperature increases and that at very low temperatures the accuracy is very low. This is caused by the Boltzman factor, which is low when the T is low. Thus very few potential states are accepted at a low temperature. Consequently the system 'moves slower' and takes longer to end up in the interesting states.

The results in appendix B showed that the accuracy increases as the number of samples increases. The cause for this is that the analytical solutions are based on systems with an infinite number of spins. Consequently system with more spins are closer to systems with

$N = \infty$, and their results are thus closer to those of the infinite system.

Another factor that increases the accuracy of the system is the number of samples. Firstly more samples means that the system gets more time to relax. Secondly a higher number of samples decreases the influence of outlier states. Lastly the probability of ending up in a interesting configuration increases as the number of spins that are potentially flipped increases.

As expected we do not observe a phase transition in the one-dimensional model.

5.2. Two-Dimensional Model

In line with the theory we have observed that there is a phase transition and that it is approximately critical. In the systems with few spins, d = 10, the phase transition is hardly visible in the plots. This is due to the analytical results being based on infinite systems.

Unfortunately we cannot offer an explanation for the drop in magnetization at $T = 2.1$ in the 10×10 system with 1000 samples.

6. Conclusion

Our findings have shown that even a small two-dimensional system with a relatively small number of samples is sufficient to observe a phase transition. However to get results that approximate the analytical solutions of the Ising model one needs to make sure that the system is suffieciently large. Furthermore one should allow the system a reasonable number of Monte Carlo steps to relax and take a large number of samples.

References

[1] Wei Cai. "Handout 12. Ising Model". Feb. 2011. URL: http://micro.stanford.edu/~caiwei/me334/Chap12_Ising_Model_v04.pdf.

[2] Ernst Ising. "Beitrag zur theorie des ferromagnetismus". In: *Zeitschrift für Physik A Hadrons and Nuclei* 31.1 (1925), pp. 253–258.

[3] Wolfgang Kenzel et al. *Physics by computer*. Springer-Verlag New York, Inc., 1997.

[4] Dirk P Kroese et al. "Why the Monte Carlo method is so important today". In: *Wiley Interdisciplinary Reviews: Computational Statistics* 6.6 (2014), pp. 386–392.

[5] David P Landau and Kurt Binder. *A guide to Monte Carlo simulations in statistical physics*. Cambridge university press, 2014.

[6] Lambert Murray. ""Classical" Statistical Physics and the Partition Function". URL: http://www.harding.edu/lmurray/themo_files/notes/ch12.pdf.

[7] Lars Onsager. "Crystal statistics. I. A two-dimensional model with an order-disorder transition". In: *Physical Review* 65.3-4 (1944), p. 117.

[8] Steven H Strogatz. *Nonlinear dynamics and chaos: with applications to physics, biology, chemistry, and engineering*. Westview press, 2014.

[9] Justin Wark. "The Partition Function of a System". URL: http://www.physics.dcu.ie/~jpm/PS302/Lect3.pdf.

[10] Wolfram Research, Inc. *Mathematica 8.0*. Version 0.8. 2010. URL: https://www.wolfram.com.

A. Mathematical Derivations

A.1. Average Energy[1]

$$
\begin{aligned}
U = & -\frac{\partial \ln[Z]}{\partial \beta} \\
= & \{\textit{Definition of } Z \textit{ in equation } (4).\} \\
& -\frac{\partial \ln\left[(2\cosh\beta)^N\right]}{\partial \beta} \\
= & \left\{\textit{Chain rule: } \frac{\partial}{\partial\beta}\ln\left[2^N\cosh^N(\beta)\right] = \frac{\partial \ln[u]}{\partial u}0,\ u = 2^n\cosh^n(\beta),\ \frac{\partial}{\partial u}\ln[u] = \frac{1}{u}\right\} \\
& -2^{-N}\cosh^{-N}(\beta)\left(\frac{\partial}{\partial\beta}\left(2^N\cosh^N(\beta)\right)\right) \\
= & \{\textit{Factor out constants.}\} \\
& -2^{-N}\frac{\partial}{\partial\beta}\left(\cosh^N(\beta)\right)2^N\cosh^{-N}(\beta) \\
= & \{\textit{Simplify the expression.}\} \\
& -\cosh(\beta)^{-N}\left(\frac{\partial}{\partial\beta}\cosh^N(\beta)\right) \\
= & \left\{\textit{Chain rule: } \frac{\partial}{\partial\beta}\cosh^N(\beta) = \frac{\partial u^N}{\partial u}0,\ u = \cosh(\beta),\ \frac{\partial}{\partial u}\left(u^N\right) = N\cdot u^{-1+N}\right\} \\
& -N\cosh(\beta)^{N-1}\frac{\partial}{\partial\beta}\left(\cosh(\beta)\right)\cosh^{-N}(\beta) \\
= & \{\textit{Simplify the expression.}\} \\
& -N\left(\frac{\partial}{\partial\beta}\cosh(\beta)\right)\operatorname{sech}(\beta) \\
= & \{\textit{Derivative of } \cosh(\alpha) \textit{ is } \sinh(\alpha).\} \\
& -\sinh(\beta)N\operatorname{sech}(\beta) \\
= & \{\textit{Simplify the expression.}\} \\
& -N\tanh(\beta)
\end{aligned}
$$

A.2. Specific Heat

$$
\begin{aligned}
C = & \frac{\partial U}{\partial T} \\
= & \{\textit{Definition of specific heat.}\} \\
& \frac{\partial U}{\partial\beta}\cdot\frac{1}{\partial T} \\
= & \{\textit{Derivate of } T \textit{ w.r.t. to } \beta.\}
\end{aligned}
$$

[1] The derivation has been computed with Wolfram Research, Inc. [10].

$$
\begin{aligned}
&\frac{\partial U}{\partial \beta} \cdot \frac{1}{{}^{-1}/_{\beta^2}} \\
=& \{Rewrite.\} \\
&-\beta^2 \frac{\partial U}{\partial \beta} \\
=& \{Definition\ of\ U.\} \\
&-\beta^2 \left(\frac{\partial}{\partial \beta} - N \tanh(\beta) \right) \\
=& \left\{ \frac{\partial}{\partial \beta} - N \tanh(\beta) = -N \frac{\partial}{\partial \beta} \tanh(\beta) = -N \operatorname{sech}^2(\beta) \right\} \\
&\beta^2 N \operatorname{sech}^2(\beta) \\
=& \left\{ Definition\ of\ \operatorname{sech}\text{:}\ \operatorname{sech}(\alpha) = \frac{1}{\cosh(\alpha)}. \right\} \\
&N \left(\frac{\beta}{\cosh(\beta)} \right)^2
\end{aligned}
$$

B. Results

Table 2: *Results of the 1D simulation for* N = *10,* $N_{samples}$ = *1000.*

		Numerical		Analytical		
T	β	${}^{U}/_{\mathrm{N}}$	${}^{C}/_{\mathrm{N}}$	${}^{U}/_{\mathrm{N}}$	${}^{C}/_{\mathrm{N}}$	accuracy
0.20	5	−0.9	0.	−1.0	0.005	$-\infty$
0.40	3	−0.9	0.003	−1.0	0.2	−32
0.60	2	−0.8	0.4	−0.9	0.4	0.88
0.80	1	−0.8	0.5	−0.8	0.4	0.94
1.0	1	−0.7	0.4	−0.8	0.4	0.96
1.2	0.8	−0.6	0.3	−0.7	0.4	0.78
1.4	0.7	−0.5	0.2	−0.6	0.3	0.77
1.6	0.6	−0.5	0.2	−0.6	0.3	0.80
1.8	0.6	−0.4	0.2	−0.5	0.2	0.88
2.0	0.5	−0.4	0.2	−0.5	0.2	0.96
2.2	0.5	−0.4	0.1	−0.4	0.2	0.81
2.4	0.4	−0.3	0.1	−0.4	0.1	0.86
2.6	0.4	−0.3	0.1	−0.4	0.1	0.91
2.8	0.4	−0.3	0.1	−0.3	0.1	0.92
3.0	0.3	−0.3	0.08	−0.3	0.10	0.84
3.2	0.3	−0.3	0.09	−0.3	0.09	0.96
3.4	0.3	−0.3	0.08	−0.3	0.08	0.97
3.6	0.3	−0.2	0.06	−0.3	0.07	0.82
3.8	0.3	−0.2	0.06	−0.3	0.06	0.86
4.0	0.3	−0.2	0.05	−0.2	0.06	0.83

Table 2: *continued*

		Numerical		Analytical		
T	β	U/N	C/N	U/N	C/N	accuracy

Table 3: *Results of the 1D simulation for* $\mathrm{N} = 10$, $N_{samples} = 10000$.

		Numerical		Analytical		
T	β	U/N	C/N	U/N	C/N	accuracy
0.20	5	-0.9	0.	-1.0	0.005	$-\infty$
0.40	3	-0.9	0.4	-1.0	0.2	0.66
0.60	2	-0.8	0.4	-0.9	0.4	0.90
0.80	1	-0.8	0.4	-0.8	0.4	0.89
1.0	1	-0.7	0.4	-0.8	0.4	0.90
1.2	0.8	-0.6	0.3	-0.7	0.4	0.88
1.4	0.7	-0.6	0.3	-0.6	0.3	0.92
1.6	0.6	-0.5	0.2	-0.6	0.3	0.87
1.8	0.6	-0.4	0.2	-0.5	0.2	0.86
2.0	0.5	-0.4	0.2	-0.5	0.2	0.88
2.2	0.5	-0.4	0.1	-0.4	0.2	0.84
2.4	0.4	-0.4	0.1	-0.4	0.1	0.92
2.6	0.4	-0.3	0.1	-0.4	0.1	0.84
2.8	0.4	-0.3	0.10	-0.3	0.1	0.88
3.0	0.3	-0.3	0.09	-0.3	0.10	0.88
3.2	0.3	-0.3	0.08	-0.3	0.09	0.88
3.4	0.3	-0.3	0.07	-0.3	0.08	0.90
3.6	0.3	-0.3	0.07	-0.3	0.07	0.93
3.8	0.3	-0.2	0.06	-0.3	0.06	0.89
4.0	0.3	-0.2	0.05	-0.2	0.06	0.88

Table 4: *Results of the 1D simulation for* $\mathrm{N} = 100$, $N_{samples} = 1000$.

		Numerical		Analytical		
T	β	U/N	C/N	U/N	C/N	accuracy
0.20	5	-1.0	0.	-1.0	0.005	$-\infty$
0.40	3	-1.0	0.004	-1.0	0.2	-20
0.60	2	-1.0	0.01	-0.9	0.4	-12
0.80	1	-0.9	0.10	-0.8	0.4	-0.74
1.0	1	-0.8	0.2	-0.8	0.4	0.30
1.2	0.8	-0.8	0.1	-0.7	0.4	0.16
1.4	0.7	-0.6	0.3	-0.6	0.3	0.95

Table 4: *continued*

		Numerical		Analytical		
T	β	U/N	C/N	U/N	C/N	accuracy
1.6	0.6	−0.5	0.2	−0.6	0.3	0.81
1.8	0.6	−0.5	0.08	−0.5	0.2	−0.049
2.0	0.5	−0.4	0.1	−0.5	0.2	0.69
2.2	0.5	−0.4	0.2	−0.4	0.2	0.94
2.4	0.4	−0.4	0.08	−0.4	0.1	0.60
2.6	0.4	−0.4	0.07	−0.4	0.1	0.61
2.8	0.4	−0.3	0.1	−0.3	0.1	0.92
3.0	0.3	−0.3	0.09	−0.3	0.10	0.89
3.2	0.3	−0.3	0.08	−0.3	0.09	0.93
3.4	0.3	−0.2	0.09	−0.3	0.08	0.79
3.6	0.3	−0.3	0.04	−0.3	0.07	0.58
3.8	0.3	−0.3	0.06	−0.3	0.06	0.91
4.0	0.3	−0.2	0.04	−0.2	0.06	0.71

Table 5: *Results of the 1D simulation for* $\mathrm{N} = 100$, $N_{samples} = 10000$.

		Numerical		Analytical		
T	β	U/N	C/N	U/N	C/N	accuracy
0.20	5	−1.0	0.	−1.0	0.005	$-\infty$
0.40	3	−1.0	0.05	−1.0	0.2	−0.021
0.60	2	−0.9	0.1	−0.9	0.4	−0.33
0.80	1	−0.8	0.6	−0.8	0.4	0.86
1.0	1	−0.8	0.4	−0.8	0.4	0.93
1.2	0.8	−0.7	0.3	−0.7	0.4	0.87
1.4	0.7	−0.6	0.3	−0.6	0.3	0.95
1.6	0.6	−0.5	0.3	−0.6	0.3	0.96
1.8	0.6	−0.5	0.2	−0.5	0.2	0.90
2.0	0.5	−0.4	0.2	−0.5	0.2	0.93
2.2	0.5	−0.4	0.1	−0.4	0.2	0.89
2.4	0.4	−0.4	0.1	−0.4	0.1	0.92
2.6	0.4	−0.4	0.1	−0.4	0.1	0.99
2.8	0.4	−0.3	0.1	−0.3	0.1	0.99
3.0	0.3	−0.3	0.1	−0.3	0.10	0.86
3.2	0.3	−0.3	0.08	−0.3	0.09	0.93
3.4	0.3	−0.3	0.08	−0.3	0.08	0.98
3.6	0.3	−0.3	0.07	−0.3	0.07	0.97
3.8	0.3	−0.3	0.07	−0.3	0.06	0.93
4.0	0.3	−0.2	0.06	−0.2	0.06	0.95

Table 6: *Results of the 1D simulation for* N = *1000,* $N_{samples}$ = *1000.*

		Numerical		Analytical		
T	β	U/N	C/N	U/N	C/N	accuracy
0.20	5	−1.0	0.	−1.0	0.005	$-\infty$
0.40	3	−1.0	0.	−1.0	0.2	$-\infty$
0.60	2	−1.0	0.	−0.9	0.4	$-\infty$
0.80	1	−1.0	0.1	−0.8	0.4	−0.72
1.0	1	−1.0	0.3	−0.8	0.4	0.65
1.2	0.8	−0.9	0.4	−0.7	0.4	0.87
1.4	0.7	−0.8	0.2	−0.6	0.3	0.62
1.6	0.6	−0.7	0.4	−0.6	0.3	0.73
1.8	0.6	−0.6	0.2	−0.5	0.2	0.70
2.0	0.5	−0.5	0.1	−0.5	0.2	0.69
2.2	0.5	−0.5	0.04	−0.4	0.2	−0.56
2.4	0.4	−0.4	0.1	−0.4	0.1	0.82
2.6	0.4	−0.4	0.03	−0.4	0.1	−0.37
2.8	0.4	−0.4	0.04	−0.3	0.1	−0.11
3.0	0.3	−0.4	0.03	−0.3	0.10	−0.23
3.2	0.3	−0.4	0.01	−0.3	0.09	−1.7
3.4	0.3	−0.3	0.02	−0.3	0.08	−0.16
3.6	0.3	−0.3	0.01	−0.3	0.07	−1.4
3.8	0.3	−0.2	0.01	−0.3	0.06	−1.4
4.0	0.3	−0.2	0.02	−0.2	0.06	−0.025

Table 7: *Results of the 1D simulation for* N = *1000,* $N_{samples}$ = *10000.*

		Numerical		Analytical		
T	β	U/N	C/N	U/N	C/N	accuracy
0.20	5	−1.0	0.	−1.0	0.005	$-\infty$
0.40	3	−1.0	0.	−1.0	0.2	$-\infty$
0.60	2	−1.0	0.07	−0.9	0.4	−1.1
0.80	1	−0.9	0.5	−0.8	0.4	0.91
1.0	1	−0.8	0.9	−0.8	0.4	0.70
1.2	0.8	−0.7	0.2	−0.7	0.4	0.54
1.4	0.7	−0.6	0.2	−0.6	0.3	0.57
1.6	0.6	−0.5	0.4	−0.6	0.3	0.80
1.8	0.6	−0.5	0.2	−0.5	0.2	0.97
2.0	0.5	−0.5	0.2	−0.5	0.2	0.97
2.2	0.5	−0.4	0.2	−0.4	0.2	0.95
2.4	0.4	−0.4	0.2	−0.4	0.1	0.90
2.6	0.4	−0.4	0.10	−0.4	0.1	0.85
2.8	0.4	−0.4	0.07	−0.3	0.1	0.71

Table 7: *continued*

		Numerical		Analytical		
T	β	U/N	C/N	U/N	C/N	accuracy
3.0	0.3	−0.3	0.07	−0.3	0.10	0.74
3.2	0.3	−0.3	0.09	−0.3	0.09	0.98
3.4	0.3	−0.3	0.07	−0.3	0.08	0.92
3.6	0.3	−0.3	0.08	−0.3	0.07	0.96
3.8	0.3	−0.3	0.05	−0.3	0.06	0.90
4.0	0.3	−0.2	0.09	−0.2	0.06	0.79

C. Implementation

C.1. Simulation

Listing 1: *./../code/MMCIsing.m*

```
function [ configurations ] = MMCIsing( initialConfiguration, parameters )
%METRPOLOISMONTECARLOISING Solve the Ising model with the MMC method.
%   InitialConfiguration is the initial configuration of the model,
%   parameters contains the parameters used in the simulation.

relaxedConfiguration = relaxSystem(initialConfiguration, parameters);
configurations = sampleSystem(relaxedConfiguration, parameters);

end

function [configuration] = relaxSystem(initialConfiguration, parameters)
    configuration = initialConfiguration;

    for i = 1:parameters.numRelaxIterations
        configuration = monteCarloStep(configuration, parameters);
    end
end

function [configurations] = sampleSystem(configuration, parameters)
    configurations = nan([size(configuration), parameters.numSampleIterations]);

    % Store the initial configuration
    configurations(:,:,1) = configuration;

    for i = 1:parameters.numSampleIterations
       configuration = monteCarloStep(configuration, parameters);
       configurations(:,:, i + 1) = configuration;
    end
end

function [nextConfig] = monteCarloStep(curConfig, parameters)
    [potConfig, flippedSpinIdx] = flipRandomSpin(curConfig);
    nextConfig = selectNextConfig(curConfig, potConfig,...
        flippedSpinIdx, parameters);
end
```

```
function [potentialConfig, flippedSpinIdx] = flipRandomSpin(currentConfig)
    potentialConfig = currentConfig;
    flippedSpinIdx = randi(numel(potentialConfig));
    potentialConfig(flippedSpinIdx) = potentialConfig(flippedSpinIdx) * -1;
end

function [nextConfig] = selectNextConfig(currentConfig, potentialConfig,
    flippedSpinIdx, parameters)
    deltaE = computeDeltaE(potentialConfig, flippedSpinIdx, parameters);
    xi = exp(- (1 / parameters.temperature) * deltaE);
    theta = rand();
    if xi > theta
        nextConfig = potentialConfig;
    else
        nextConfig = currentConfig;
    end
end

function [deltaE] = computeDeltaE(potentialConfig, flippedSpinIdx, parameters)
    neighbors = parameters.neighborFunction(flippedSpinIdx, potentialConfig);
    deltaE = -2 * potentialConfig(flippedSpinIdx) * sum(neighbors);
end
```

Listing 2: *./../code/+neighbors/OneD2Connected.m*

```
function [neighbors] = OneD2Connected( elementIdx, matrix )
%ONED2CONNECTED The 2-connected neighbours of ELEMENT in the 1D MATRIX.

if ~isvector(matrix)
   error('Error. \nMatrix must be 1 dimensional not %d-dimensional.',...
       length(size(matrix)))
end

    left_neighbor = [];
    if elementIdx > 1
        left_idx = elementIdx - 1;
        left_neighbor = matrix(left_idx);
    end

    right_neighbor = [];
    if elementIdx < length(matrix)
        right_idx = elementIdx + 1;
        right_neighbor = matrix(right_idx);
    end

    neighbors = [left_neighbor, right_neighbor];
end
```

Listing 3: *./../code/+neighbors/TwoD4Connected.m*

```
function [ neighbors ] = TwoD4Connected( elementIdx, matrix )
%TWOD4CONNECTED The 2-connected neighbours of ELEMENT in  the 2D MATRIX.

    if length(size(matrix)) ~= 2
       error('Error. \nMatrix must be 2 dimensional not %d-dimensional.',...
           length(size(matrix)))
```

```
    end

    [row, col] = ind2sub(size(matrix), elementIdx);

    left = left_neighbor(row, col, matrix);
    right = right_neighbor(row, col, matrix);
    top = top_neighbor(row, col, matrix);
    bottom = bottom_neighbor(row, col, matrix);

    neighbors = [left, right, top, bottom];
end

function[left] = left_neighbor(row, col, matrix)
    left = [];
    if col > 1
        left_idx = col - 1;
        left = matrix(row, left_idx);
    end
end

function[right] = right_neighbor(row, col, matrix)
    right = [];
    if col < size(matrix, 2)
        right_idx = col + 1;
        right = matrix(row, right_idx);
    end
end

function [top] = top_neighbor(row, col, matrix)
    top = [];
    if row > 1
        top_idx = row - 1;
        top = matrix(top_idx, col);
    end
end

function [bottom] = bottom_neighbor(row, col, matrix)
    bottom = [];
    if row < size(matrix, 1)
        bottom_idx = row + 1;
        bottom = matrix(bottom_idx, col);
    end
end
```

C.2. Statistics

Listing 4: *./../code/computeAverageEnergy.m*

```
function [ average, energies ] = computeAverageEnergy( configurations )
%COMPUTEAVERAGEENERGY Compute the average energy of a list of CONFIGURATIONS.
% Energies is the list of energies that the average is based on.

    energies = computeEnergies(configurations);
    average = mean(energies);
end
```

```
function [energies] = computeEnergies(configurations)
    numConfigurations = size(configurations, 3);

    extra_column = zeros(size(configurations, 1), 1, numConfigurations);
    shifted_left = [extra_column, configurations];
    shifted_right = [configurations, extra_column];
    col_term = shifted_left .* shifted_right;
    col_term_sum = sum(sum(col_term, 1), 2);

    % This term is always zero for 1D models.
    extra_row = zeros(1, size(configurations, 2), numConfigurations);
    shifted_top = [extra_row; configurations];
    shifted_down = [configurations; extra_row];
    row_term = shifted_down .* shifted_top;
    row_term_sum = sum(sum(row_term, 1), 2);

    energies = - col_term_sum - row_term_sum;
end
```

Listing 5: *./../code/computeSpecificHeat.m*

```
function [ specificHeat ] = computeSpecificHeat( configurations, temperature)
%COMPUTESPECIFICHEAT Compute the specific heat of a set of configurations.
    [U, energies] = computeAverageEnergy(configurations);
    beta = (1 / temperature);
    [~, ~, number_of_configurations] = size(configurations);
    specificHeat = beta^2 * ( (1/number_of_configurations) * sum(energies.^2)  -
    U.^2);
end
```

Listing 6: *./../code/computeAccuracy.m*

```
function [ accuracy ] = computeAccuracy( actual, theory )
%ACCURACY Compute the accuracy of the ACTUAL values w.r.t. the value in THEORY.
accuracy = ones(size(actual)) - abs((abs(actual - theory) ./ actual));
end
```

Listing 7: *./../code/computeAverageMagnetization.m*

```
function [ average ] = computeAverageMagnetization( configurations )
%COMPUTEAVERAGEMAGNETIZATION Compute average magnetization of a set of
    CONFIGURATIONS.
    magnetizations = sum(sum(configurations, 1), 2);
    average= mean(magnetizations);
end
```

C.3. Experiments

Listing 8: *./../code/experiment_1D.m*

```
clc; close all; clear all;
rng('default');

computeNumRelaxIterations = @(n) 1/10 .* n;
```

```
%% Init
temperatures = 0.2:0.2:4;
numParticlesList = [10, 100, 1000];
numSampleIterationsList = [1000, 10000];

idx = 1;

%% Run Simulations
for numParticles = numParticlesList
    for numSampleIterations =  numSampleIterationsList
        numRelaxIterations = computeNumRelaxIterations(numSampleIterations);
        initialConfiguration = ones([1, numParticles]);
        for temperature = temperatures

          parameters = struct(...
            'temperature', temperature,...
            'numParticles', numParticles,...
            'numSampleIterations', numSampleIterations,...
            'numRelaxIterations', numRelaxIterations,...
            'neighborFunction', @neighbors.OneD2Connected);

          configurations = MMCIsing(initialConfiguration, parameters);

          % The initial configuration for the next temperature is the
          % final configuration of the previous temperature
          initialConfiguration = configurations(:,:,end);

          parameters = rmfield(parameters, 'neighborFunction');

          experiments(idx).parameters = parameters;
          experiments(idx).configurations = configurations;

          U = computeAverageEnergy(configurations);
          C = computeSpecificHeat(configurations, parameters.temperature);

          experiments(idx).statistics = struct('averageEnergy', U,...
              'specificHeat', C);
          idx = idx + 1;
      end
   end
end

%% Store the results
save('../results/1D.mat', 'experiments')
```

Listing 9: *./../code/experiment_2D.m*

```
clc; close all; clear all;
rng('default');

computeNumRelaxIterations = @(n) 1/10 .* n;

%% Init
temperatures = 0.2:0.2:4;
dimensionalityList = [10, 50, 100];
numSampleIterationsList = [1000, 10000];

idx = 1;
```

```
%% Run Simulations
for dimensionality = dimensionalityList
    numParticles = dimensionality * dimensionality;

    for numSampleIterations =  numSampleIterationsList
        numRelaxIterations = computeNumRelaxIterations(numSampleIterations);
        initialConfiguration = ones([dimensionality, dimensionality]);

        for temperature = temperatures

          parameters = struct(...
            'temperature', temperature,...
            'numParticles', numParticles,...
            'numSampleIterations', numSampleIterations,...
            'numRelaxIterations', numRelaxIterations,...
            'neighborFunction', @neighbors.TwoD4Connected);

          configurations = MMCIsing(initialConfiguration, parameters);

          % The initial configuration for the next temperature is the
          % final configuration of the previous temperature
          initialConfiguration = configurations(:,:,end);

          parameters = rmfield(parameters, 'neighborFunction');

          experiments(idx).parameters = parameters;
%           experiments(idx).configurations = configurations;

          U = computeAverageEnergy(configurations);
          C = computeSpecificHeat(configurations, parameters.temperature);
          M = computeAverageMagnetization(configurations);

          experiments(idx).statistics = struct(...
              'averageEnergy', U,...
              'specificHeat', C,...
              'averageMagnetization', M);
          idx = idx + 1;

          clear configurations
        end
    end
end

%% Store the results
save('../results/2D.mat', 'experiments')

%% Create plots
output.plot(experiments, 'averageEnergy');
output.plot(experiments, 'specificHeat');
output.plot(experiments, 'averageMagnetization',...
    'theoretical', @theory.averageMagnetizationPerSpin);
```

C.4. Output

Listing 10: *./../code/tables_1D.m*

```
clc; close all; clear all;
load('../results/1D.mat');

numParticlesList = [10, 100, 1000];
numSampleIterationsList = [1000, 10000];

   for numParticles = numParticlesList
      for numSampleIterations =  numSampleIterationsList
        accuracy = output.generateLaTeXTable(experiments, numSampleIterations,
    numParticles);
        fprintf('%d %d %f\n', numParticles, numSampleIterations, accuracy)
      end
      fprintf('\n')
   end
```

Listing 11: *./../code/+output/generateLaTeXTable.m*

```
function [mean_accuracy] = generateLaTeXTable( experiments, numSampleIterations,
    numParticles)
%GENERATELATEXTABLE Generate a LaTeX table for the experiments.
    data = collectData(experiments, numSampleIterations, numParticles);

    mean_accuracy = meanAccuracy(data);

    outputFile = generatePath(numSampleIterations, numParticles);

    fileID = fopen(outputFile,'w+');
    fprintf(fileID, '%%!TEX root = ../report.tex\n\n');
    fprintf(fileID, '\\num[round-precision=2]{%1.1f}\t& \\num{%0.5f}\t& \\num
    {%0.5f}\t& \\num{%0.5f}\t& \\num{%0.5f}\t& \\num{%0.5f}\t& \\num[round-
    precision=2]{%0.5f}\t\\\\\n', data);
    fclose(fileID);

end

function [outputFilePath] = generatePath(numSampleIterations, numParticles)
    outputFilePath = sprintf('../report/tables/NS%d_N%d.tex', ...
        numSampleIterations, numParticles);
end

function [average] = meanAccuracy(data)
    accuracies = data(end, :);
    average = mean(accuracies(~isinf(accuracies)));
end
```

Listing 12: *./../code/+output/plot.m*

```
function [figure] = plot( experiments, statistic, varargin)
%PLOT the parameter per spin as a function of temperature.
    parser = inputParser;
    parser.addRequired('experiments');
    parser.addRequired('statistic');
    addParameter(parser, 'theoretical', nan);
    parser.parse(experiments, statistic, varargin{:});

    parameters = [experiments.parameters];
```

```
    numParticlesList = unique([parameters.numParticles]);
    numSampleIterationsList = unique([parameters.numSampleIterations]);

    % source: https://plot.ly/ipython-notebooks/color-scales/
    colors = {'#AECDE1', '#3C76AF',...
        '#BBDE93', '#559E3F', ...
        '#ED9F9C', '#D0352C', ...
        '#F4C17B', '#EF8632'};

    i = 1;
    for numParticles = numParticlesList
        for numSampleIterations = numSampleIterationsList
            traces{i} = createPlotStruct(experiments, statistic, numParticles,
    numSampleIterations, colors{i});
            i = i + 1;
        end
    end

    if isa(parser.Results.theoretical, 'function_handle')
        traces{i} = createTheoreticalPlotStruct(parameters, parser.Results.
    theoretical, colors{i});
    end

    figure = createPlot(traces, statistic);
    writePlot(figure, statistic);
end

function [] = writePlot(figure, statistic)
    outputFile = sprintf('../report/img/2D/%s.pdf', statistic);
    saveplotlyfig(figure, outputFile);
end

function [figure] = createPlot(traces, statistic)
    yLabel = createYLabel(statistic);
    layout = struct(...
        'legend', struct(...
            'orientation', 'h'),...
        'font', struct('size', 18),...
        'xaxis', struct(...
            'title', '$T$',...
            'autotick', false, ...
            'tick0', 0,...
            'dtick', 0.4,...
            'tickangle', 0),...
        'yaxis', struct(...
            'title', yLabel)...
        );
    figure.data = traces;
    figure.layout = layout;
    figure.UserData = struct(...
        'filename', 'latex',...
        'fileopt', 'overwrite');
    figure.UserData = struct('filename', 'latex', 'fileopt', 'overwrite');
end

function [yLabel] = createYLabel(statistic)
    if strcmp(statistic, 'averageEnergy')
```

```
        statisticStr = 'U';
    elseif strcmp(statistic,'specificHeat')
        statisticStr = 'C';
    elseif strcmp(statistic, 'averageMagnetization')
        statisticStr = 'M';
    end
    yLabel= sprintf('$\\frac{%s}{N}$', statisticStr);
end

function [trace] = createPlotStruct(experiments, statistic, numParticles,
    numSampleIterations, color)
    [temperatures, dependent] = getData(experiments, statistic, numParticles,
    numSampleIterations);
    dimensionality = sqrt(numParticles);
    trace = struct(...
          'x', temperatures, ...
          'y', dependent, ...
          'name', sprintf('$d = %03d, N_\\textit{samples} = %1.0g$',
    dimensionality, numSampleIterations), ...
          'mode', 'lines+markers' ,...
          'line', struct(...
                'color', color,...
                'dash', 'solid',...
                'width', 2),...
          'type', 'scatter');
end

function [trace] = createTheoreticalPlotStruct(parameters, theoretical, color)
    temperatures = unique([parameters.temperature]);
    temperatures = linspace(min(temperatures), max(temperatures), 100);
    dependent = theoretical(temperatures);
    trace = struct(...
          'x', temperatures, ...
          'y', dependent, ...
          'name', sprintf('theoretical'), ...
          'mode', 'lines' ,...
          'type', 'scatter',...
          'line', struct(...
                'dash', 'dashdot',...
                'width', 2,...
                'color', color)...
            );
end

function [temperatures, dependent] = getData(experiments, statistic, numParticles,
     numSampleIterations)
    subset = filterExperiments(experiments,...
        'numParticles', numParticles,...
        'numSampleIterations', numSampleIterations...
    );

    dependent = getStatistic(subset, statistic) ./ numParticles;
    temperatures = getTemperatures(subset);
end

function [dependent] = getStatistic(subset, statistic)
    statistics = [subset.statistics];
```

```
    dependent = nan(1, length(statistics));
    for i = 1:length(statistics)
        dependent(i) = getfield(statistics(i), statistic);
    end
end

function [temperatures] = getTemperatures(subset)
    parameters = [subset.parameters];
    temperatures = [parameters.temperature];
end
```

Listing 13: *./../code/+output/private/filterExperiments.m*

```
function [ subset ] = filterExperiments( experiments, varargin )
%FILTERCONFIGURATIONS Filter configurations on different parameters.

    parser = inputParser;
    parser.addRequired('experiments');
    addParameter(parser, 'temperature', nan, @isnumeric);
    addParameter(parser, 'numParticles', nan, @isnumeric);
    addParameter(parser, 'numSampleIterations', nan, @isnumeric);
    addParameter(parser, 'numRelaxIterations', nan, @isnumeric);

    parser.parse(experiments, varargin{:});

    subset = experiments;

    if ~ isnan(parser.Results.temperature)
        parameters = [subset.parameters];
        subset = subset([parameters.temperature] ...
            == parser.Results.temperature);
    end

    if ~ isnan(parser.Results.numParticles)
        parameters = [subset.parameters];
        subset = subset([parameters.numParticles] ...
            == parser.Results.numParticles);
    end

    if ~ isnan(parser.Results.numSampleIterations)
        parameters = [subset.parameters];
        subset = subset([parameters.numSampleIterations] ...
            == parser.Results.numSampleIterations);
    end

    if ~ isnan(parser.Results.numRelaxIterations)
        parameters = [subset.parameters];
        subset = subset([parameters.numRelaxIterations] ...
            == parser.Results.numRelaxIterations);
    end

end
```

Listing 14: *./../code/+output/private/collectData.m*

```
function [data] = collectData(experiments, numSampleIterations, numParticles)
    subset = filterExperiments(experiments,...
```

```
        'numSampleIterations', numSampleIterations,...
        'numParticles', numParticles);

    parameters = [subset.parameters];
    statistics = [subset.statistics];

    temperature = [parameters.temperature];
    beta = 1 ./ temperature;

    UperSpin_MMC = [statistics.averageEnergy]./numParticles;
    CperSpin_MMC = [statistics.specificHeat]./numParticles;

    UperSpin_theory = theory.averageEnergyPerSpin1D(temperature);
    CperSpin_theory = theory.specificHeatPerSpin1D(temperature);

    accuracy = mean([...
       computeAccuracy(UperSpin_MMC, UperSpin_theory);...
       computeAccuracy(CperSpin_MMC, CperSpin_theory)]);

   data = [temperature; beta;...
       UperSpin_MMC; CperSpin_MMC;...
       UperSpin_theory; CperSpin_theory; accuracy];
end
```