One- and Two-Dimensional Ising Model

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

September 13, 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, 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 estimate 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, s = +1, or down, s = -1 [8]. The magnetization of a magnet is defined as its average spin:

$$\mathrm{M} = \left| \frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} \mathrm{s}_i \right|,$$

where N is the number of spins. 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 op 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

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

Where $\langle i, j \rangle$ is a nearest neighbour pair, the nearest neighbour 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 particles. In a ferromagnetic model, $\mathcal{J} > 0$ neighboring spins prefer to be parallel. In the anti-ferromagnetic model, $\mathcal{J} < 0$ spins prefer a direction different to one of their neighbors. The constant h represents the external magnetic field, the spins want to align with the direction of h, i.e. when $h > 0$ 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

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

*Master Profile: Computing Science
Student Number: s1869140

The probability of a configuration of spins $\mathcal{S}$ at temperature T is given by

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

where $T = {}^{1}/_{\beta}$ and Z is the partition function:

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

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

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

Wat is E_n?

Waarover loopt die som?

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

$$U = \frac{1}{Z} \cdot \sum_n E_n \cdot \exp\left[-\beta E_n\right].$$

Observing that

$$\frac{\partial Z}{\partial \beta} = \sum_n -E_n \exp\left[-\beta E_n\right],$$

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

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

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], consequently

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

1.1.2. Two-Dimensional Model

The 2D Ising model is a square lattice whose lattice sites are occupied by spins. Each spin has either a positive or a negative spin [3]. The Hamiltonian of the 2D model is the same as the one of the one dimensional model given in equation (1). The pairs of nearest neighbours are now found by looking at the four connected neighbours, i.e. the nearest neighbours 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 $\mathrm{d} \times \mathrm{d}$ spins is computed as

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

Wat als ie op de rand zit, lopen we rond, of zien we gewoon minder?

The two-dimensional Ising model has been solved analytically by Onsager [7]. He showed that the average magnetization per spin on a infinite 2D 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 equilibirium 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 this complexity problem.

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. Monte Carlo experiments can be used for 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 randomly generated 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].

The Metropolis Monte Carlo 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.

Section 2 discusses how the Metropolis Monte Carlo method was 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,` $\mathcal{S}_{cur}$`)`
  $\mathcal{S}_{cur}$:= `selectConfig(` $\mathcal{S}_{cur}$`,` $\mathcal{S}_{pot}$
  `)`

Algorithm 1 requires an initial configuration $\mathcal{S}_{init}$ as input, this configuration is a representation of the system in its initial state. The parameter $N_{iterations}$ indicates how many configurations are generated, generally $N_{iterations} = \mathrm{N}$. 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 `selectConfig()`. The pseudo code of this function is presented in algorithm 2.

Given two configurations `selectConfig()` selects with which one the simulation should continue. To this end ΔE, the difference in energy between

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

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 $\mathcal{S}_{new} := \mathcal{S}_{cur}$

the two configurations, is computed. It is not possible to compute the energy of the two configurations according to equation (5). However writing equation (1) as

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

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:

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

If a the potential configuration is determined is decided by two values: θ and ξ. The first is sampled from a pseudo random uniform distribution with the range $(0, 1)$. The second is computed from ΔE according to

$$\xi = \exp\left[-\beta \Delta E\right]. \tag{10}$$

Formuleer opnieuw, dit loopt niet helemaal lekker.

If the transition from $\mathcal{S}_{cur}$ to $\mathcal{S}_{pot}$ decreases the energy of the system ξ is greater than one and as $\theta \in (0, 1)$ this means that potential states with a lower energy than 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 appendix B.

Refer to exact listings.

3. Experiments

This section introduces the experiments we ran with the model introduced in the previous sections. In the experiments below the number of iterations, $N_{iterations}$, is not necessarily equal to N, the number of spins, but set independently. 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 1D 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

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

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

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

Furthermore we will compare the results of the simulation with the analytical solution presented earlier.

Refer to listings/section in appendix with the code for this experiment.

3.2. Two-Dimensional Model

In the 2D model we are not only interested in the average energy and specific heat per spin but also the average magnetization per spin. The following parameter space is used: $T = 0.2, 0.4, \ldots, 4$, N = 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).

Formule toevoegen?

Refer to listings/section in appendix with the code for this experiment.

4. Results

Wat gaan we doen?

4.1. One-Dimensional Model

4.1.1. Average Energy

Report average energy for different values of T, N and NSAMPLES

4.2. Specific Heat

Report specific heat for different values of T, N and NSAMPLES

4.3. Two-Dimensional Model

Wat gaan we doen?

Average Energy

Report average energy for different values of T, N and NSAMPLES

Specific Heat

Report specific heat for different values of T, N and NSAMPLES

Average Magnetization

Report average magnetization for different values of T, N and NSAMPLES

5. Discussion

What are we going to discuss?

Interpret results in terms of a phase transition from a state with magnetization zero to a state with definite magnetization (slide 31)

Invloed van de parameters, T, N, NSAMPLES p 192 physics by computer

5.1. One-Dimensional Model

Compare results with the analytical solution

5.2. Two-Dimensional Model

Compare Average magnetization with the exact result for the infinite system

6. Conclusion

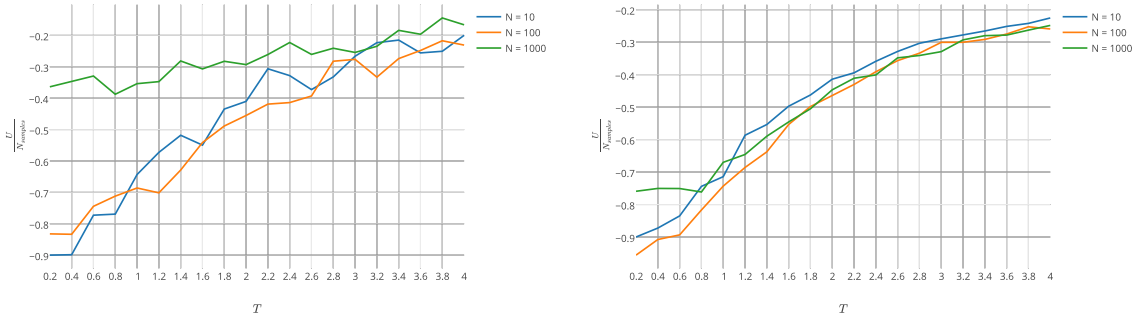
Hoe goed sluit het model aan bij de het exacte resultaat?

Wat hebben we geleerd over de parameters.

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.



(a) $N_{iterations} = 1000$ **(b)** $N_{iterations} = 10000$

Figure 1: *The average energy for (a) 1000 and (b) sample iterations.*

[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].

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

B. Implementation

B.1. Simulation

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

```
function [ configurations ] = metropolisMonteCarloIsing(
   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(currentConfig, parameters)
    [potentialConfig, flippedSpinIdx] = flipRandomSpin(currentConfig);
    nextConfig = selectNextConfig(currentConfig, potentialConfig,
   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/generateRandomConfiguration.m*

```
function [ configuration ] = generateRandomConfiguration(
   configurationSize )
%GENERATERANDOMCONFIGURATION Generate a random configuation of particles.
%   Particles are either positive (+1) or negative (-1). Size is an array
%   with the size of each dimension of the configuration.

configuration = round(rand(configurationSize));

% Map from [0, 1] to [-1, 1]
configuration = -1 + 2 * configuration;
```

```
end
```

B.2. Statistics

Listing 3: *./../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.

    [dimension, ~, ~] = size(configurations);
    if dimension == 1
        [average, energies] = averageEnergy1D(configurations);
    else
        [average, energies] = averageEnergy2D(configurations);
    end

end

function [average, energies] = averageEnergy1D( configurations )
    [~, ~, num_configurations] = size(configurations);
    shifted_left = [zeros(1, 1, num_configurations), configurations];
    shifted_right = [configurations, zeros(1, 1, num_configurations)];
    multiplied = shifted_left .* shifted_right;
    energies = -1 * sum(multiplied, 2);
    average = mean(energies);
end

function [average, energies] = averageEnergy2D( configurations )
    error('Error. \nNot yet implemented.');
end
```

Listing 4: *./../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 5: *./../code/computeMagnetization.m*

B.3. Experiments

Listing 6: *./../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 temperature = temperatures
   for numParticles = numParticlesList
      for numSampleIterations =  numSampleIterationsList
          numRelaxIterations = computeNumRelaxIterations(
   numSampleIterations);

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

          initial_configuration = generateRandomConfiguration([1,
   numParticles]);
          configurations = metropolisMonteCarloIsing(
   initial_configuration, parameters);

          parameters = rmfield(parameters, 'neighborFunction');

          experiments(idx).parameters = parameters;
          experiments(idx).configurations = configurations;
          experiments(idx).statistics = struct(...
              'averageEnergy', computeAverageEnergy(configurations),...
              'specificHeat', computeSpecificHeat(configurations,
   parameters.temperature));
          idx = idx + 1;
      end
   end
end

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


%% Compute requested data
```

Listing 7: *./../code/experiment_2D.m*

B.4. Plots

Listing 8: *./../code/plot_1D.m*

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

%% Average Energy for N_samples = 1000, T = 0.2:4, N = 10, 100, 1000
figure = plots.averageEnergy1D(experiments, 1000);
saveplotlyfig(figure, '../report/img/1D/1DaverageEnergyN1000.pdf');

%% Average Energy for N_samples = 10000, T = 0.2:4, N = 10, 100, 1000
figure = plots.averageEnergy1D(experiments, 10000);
saveplotlyfig(figure, '../report/img/1D/1DaverageEnergyN10000.pdf');

%% Sepcific Heat for N_samples = 1000, T = 0.2:4, N = 10, 100, 1000
figure = plots.specificHeat1D(experiments, 1000);
saveplotlyfig(figure, '../report/img/1D/1DspecificHeatN1000.pdf');

%% Sepcific Heat for N_samples = 10000, T = 0.2:4, N = 10, 100, 1000
figure = plots.specificHeat1D(experiments, 10000);
saveplotlyfig(figure, '../report/img/1D/1DspecificHeatN10000.pdf');
```

Listing 9: *./../code/plot_2D.m*

Listing 10: *./../code/+plots/averageEnergy1D.m*

```
function [ figure ] = averageEnergy1D( experiments,
    numberOfSampleIterations, varargin )
%AVERAGEENERGY1D Generate the Plotly handles for the U/N plots.
%   Plots U/N as a function of T for different values of N, i.e. number
    of
%   samples.

    parser = inputParser;
    parser.addRequired('experiments');
    parser.addRequired('numberOfSampleIterations');
    parser.addParameter('numParticles', [10, 100, 1000]);
    parser.parse(experiments, numberOfSampleIterations, varargin{:});

    subset = filterExperiments(experiments, 'numSampleIterations',
    numberOfSampleIterations);

    for i = 1:length(parser.Results.numParticles)
        N = parser.Results.numParticles(i);

       subsubset = filterExperiments(subset, 'numParticles', N);
       statistics = [subsubset.statistics];
       parameters = [subsubset.parameters];
```

```
        averageEnergiesPerSpin = [statistics.averageEnergy]./N;
        temperatures = [parameters.temperature];

         traces{i} = struct(...
           'x', temperatures, ...
           'y', averageEnergiesPerSpin, ...
           'name', sprintf('N = %d', N), ...
           'mode', 'lines' ,...
           'type', 'scatter');
    end

    layout = struct(...
        'xaxis', struct('title', '$T$', 'autotick', false, 'tick0', 0, '
    dtick', 0.2),...
        'yaxis', struct('title', '$\frac{U}{N_\textit{samples}}$'));

    figure.data = traces;
    figure.layout = layout;
    figure.UserData = struct('filename', 'latex', 'fileopt', 'overwrite');

%     response = plotly(traces, struct('layout', layout, 'filename', '
    latex', 'fileopt', 'overwrite'));
end
```

Listing 11: *./../code/+plots/specificHeat1D.m*

```
function [ figure ] = specificHeat1D ( experiments,
    numberOfSampleIterations, varargin)
%SPECIFICHEAT1D Generate the Plotly handles for the C/N plots.
%   Plots U/N as a function of T for different values of N, i.e. number
    of
%   samples.

    parser = inputParser;
    parser.addRequired('experiments');
    parser.addRequired('numberOfSampleIterations');
    parser.addParameter('numParticles', [10, 100, 1000]);
    parser.parse(experiments, numberOfSampleIterations, varargin{:});

    subset = filterExperiments(experiments, 'numSampleIterations',
    numberOfSampleIterations);

    for i = 1:length(parser.Results.numParticles)
        N = parser.Results.numParticles(i);

        subsubset = filterExperiments(subset, 'numParticles', N);
        statistics = [subsubset.statistics];
        parameters = [subsubset.parameters];

        specificHeatPerSpin = [statistics.specificHeat]./N;
        temperatures = [parameters.temperature];
```

```
        traces{i} = struct(...
          'x', temperatures, ...
          'y', specificHeatPerSpin, ...
          'name', sprintf('N = %d', N), ...
          'mode', 'lines' ,...
          'type', 'scatter');
    end

    layout = struct(...
        'xaxis', struct('title', '$T$', 'autotick', false, 'tick0', 0, '
    dtick', 0.2),...
        'yaxis', struct('title', '$\frac{C}{N_\textit{samples}}$'));

    figure.data = traces;
    figure.layout = layout;
    figure.UserData = struct('filename', 'latex', 'fileopt', 'overwrite');


end
```

Listing 12: *./../code/+plots/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
```

B.5. Tables

Listing 13: *./../code/tables_1D.m*

Listing 14: *./../code/tables_2D.m*