Theoretical Practical Course in Computational Physics

Tobias Göppel and Sophia Kronthaler

October 10, 2014

Contents

1 Monte Carlo (MC) integration method

The Monte Carlo integration method is quite similar to the Riemann integration method with the subtle difference that one chooses the x_is randomly. This leads to the following approximative formula for the integral:

$$I = \frac{b-a}{N} \sum_{i=0}^{N-1} f(x_i) \xrightarrow{N\to\infty} \int_a^b f(x)\,\mathrm{dx} \tag{1}$$

We can estimate the integral of the function f by:

$$\int_a^b f(x), \mathrm{dx} \approx I \pm Error = V \langle f \rangle \pm V \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \tag{2}$$

Of course we do not know the standard deviation of f but we can approximate it "on the flight" when performing the Monte Carlo integration. In our program, we assume that the desired accuracy is reached, when:

$$accuracy \geq \frac{\sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}}{\langle f \rangle}$$

As one can see, the error decreases with $\sim \frac{1}{\sqrt{N}}$. To get an more detailed idea of what determines the error, we should notice that the individual function values at random points x_i one the x-axis are themselvs random numbers.The Integrand, being sum of random numbers is a random number too. The distribution of the integrated values approaches a Gaussian. By using the central limit theorem it can be shown that the following expression holds:

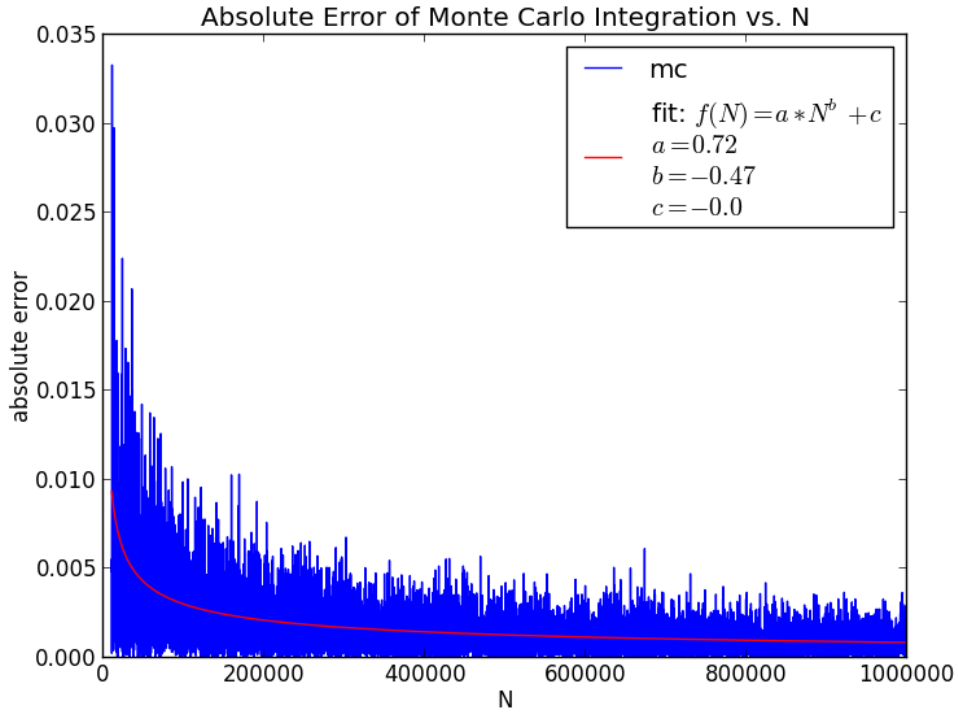
$$\sigma^2(I_N) = \frac{V^2}{N} \int_V (f(x) - \langle f \rangle)^2 \mathrm{dx} = \frac{V^2}{N} \sigma^2(f) \tag{3}$$

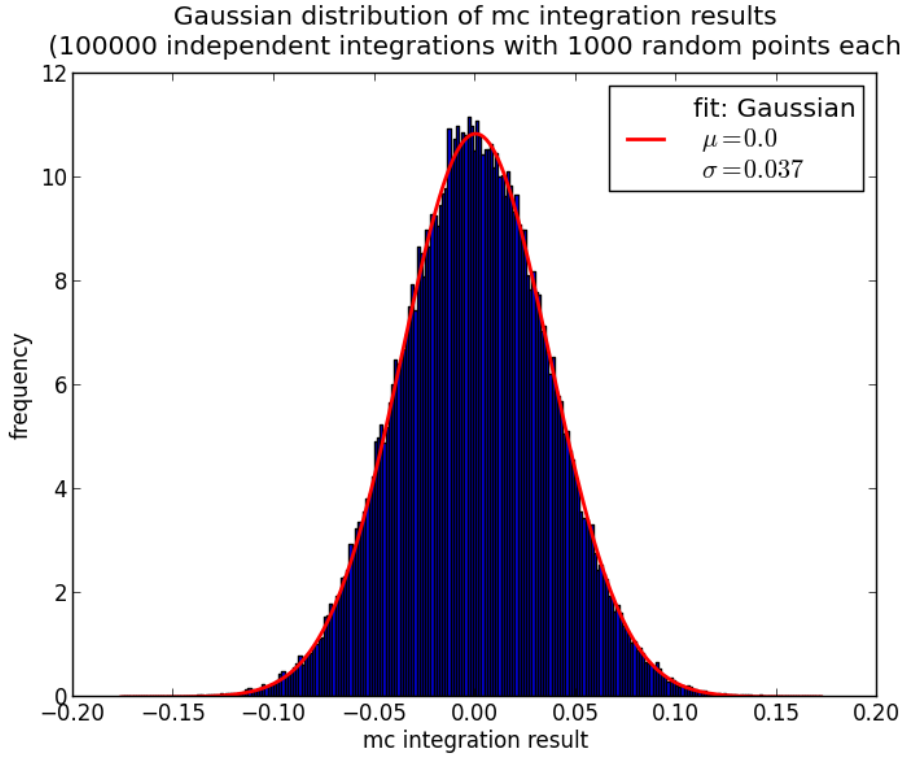
This leads to the important conclusion that the variance of I_N does not depend only on the quantity of random points N, but also on the volume V and the variance of the function $\sigma^2(f)$. To show this characteristic we define the ratio ρ.

$$\rho = \frac{N}{V^2} \frac{\sigma^2(I_N)}{\sigma^2(f)}$$

Figure **??** shows the result by integrating $f(x) = x$ via the Monte Carlo method. Furthermore results for different functions can be found in the appendix **??**.

Figure 1: $f(x) = x$, volume: $[-1, 1]$, ratio: 1.051





2 Ising Model

The Ising model is a well known toy model for ferromagnetism, for that reason it will not be explained in detail here. The Hamiltonian for a predefined spin configration $\{c\}$ is given by:

$$H(\{c\}) = -\frac{1}{2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} J_{ij}s_i s_j - B\sum_{i=0}^{N-1} s_i$$

J_{ij} (coupling term) determines the strength of the force exerted in an interaction of neighboring spins. It depends on the sign of the coupling constant whether the ground state is ordered or disordered.

$J_{ij} > 0$the interaction is called ferromagnetic with an ordered GS: $E_0 = -NJ$

$J_{ij} < 0$the interaction is called anti-ferromagnetic with an disordered GS: $E_0 = +NJ$

In this model we solely consider nearest neighbors interaction with periodic boundary conditions. With an occupation probability for a specific spin configuration of

$$P(\{c\}) \sim \exp[-\beta H(\vec{S})]\text{with } \beta = \frac{1}{k_B T}$$

one can compute the observables listed below:

1d without magnetic field (per spin):

$$\langle u\rangle = \sum_{\{c\}} H(\{c\})P(\{c\}) = -J\tanh(\beta J)$$
$$\langle c\rangle = k_B(\beta J)^2[1-\tanh^2(K)]$$
$$\langle m\rangle = 0$$

1d with magnetic field (per spin):

$$\langle m\rangle = \frac{\sinh(h)}{\sqrt{\sinh^2(h)+\exp(-4\beta J)}}$$
$$\langle \chi\rangle = \beta\cosh(h)\frac{\exp(-4J\beta)}{(\sinh(h)+\exp(-4J\beta))^{\frac{3}{2}}}$$

To obtain the exact solution for the 2d Ising Model is a story in itself.[1]

[1]We recommend the Advanced Statistical Physics Script,
Prof. Dr. Ulrich Schollwöck, Chapter 7

3 Metropolis Algorithm

The aim is to generate configurations of Ising systems in thermal equilibrium at a given temperature. The standard method, the Metropolis Algorithm, of acquiring these sample configurations is actually a modified Monte Carlo scheme 'Instead of choosing configurations randomly and then weighting them, we choose configurations with a probability $P \sim \exp(-\frac{E}{k_b T})$) and weight them evenly.'[1]
The following steps illustrate the Metropolis method:
In the beginning, select some initial spin configuration $\{c^0\}$ and compute its energy $E_0 = H(\{c^0\})$. Then:

1. Randomly select one spin s_i of the configuration $\{c\}$
2. Flip $s_i \to -s_i$ to obtain $\{c^n\} \to \{c^*\}$
3. Compute the energy $E_* = H(\{c^*\})$ for the new configuration
4. If $E_* \leq E_n$ accept the new configuration $\{c^{n+1}\} = \{c^*\}$
5. Else accept the new configuration with a probability of $\exp(-\beta \triangle E)$
6. If finally rejected duplicate old configuration $\{c^{n+1}\} = \{c^n\}$
7. Compute $E_{n+1} = H(\{c^{n+1}\})$

To speed up the simulation, we modified these steps by avoiding exponentiations and total energy evaluations inside the metropolis loop. On that account we calculate the 'acceptance-' or 'weightening-cases' in advance by hand and merely check whichever takes place. In the beginning, select some initial configuration $\{c^0\}$, compute its energy $E_0 = H(\{c^0\})$, its energy changes by $\triangle E$ by flipping a spin and its weights $P(\triangle E)$.

1. Randomly select one spin s_i of the configuration $\{c\}$
2. Check which case is applicable
3. If it's an 'acceptance-case', flip the spin and calculate the new energy by $E_{n+1} = E_n + \triangle E$
4. Else accept the new configuration with the applicable probability saved in the weights

The specific cases can be found in Table 1, 2, 3.

Table 1: 1D without magnetic field

case	$\triangle$E	weight
$+++ \rightarrow +-+$	$+8J$	$exp(-8\beta J)$
$+-+ \rightarrow +++$	$-4J$	accept
$+-- \rightarrow ++-$	0	accept

Table 2: 1D with magnetic field (h field is +)

case	$\triangle$E	weight
$+++ \rightarrow +-+$	$4J+2h$	$exp(-4\beta J-2\beta h)$
$--- \rightarrow -+-$	$4J-2h$	if $4J > 2h$: $\exp(-4\beta J+2\beta h)$ $else$: accept
$+-+ \rightarrow +++$	$-4J-2h$	accept
$-+- \rightarrow ---$	$-4J+2h$	if $4J < 2h$: $\exp(4\beta J-2\beta h)$ $else$: accept
$++- \rightarrow +--$	$2h$	$\exp(-2\beta h)$
$--+ \rightarrow -++$	$2h$	accept

Table 3: 2D without magnetic field

case	$\triangle$E	weight
$\begin{matrix}+++ \\ +++ \\ +++\end{matrix} \longrightarrow \begin{matrix}+++ \\ +-+ \\ +++\end{matrix}$	$8J$	$exp(-8\beta J)$
$\begin{matrix}+++ \\ +++ \\ ++-\end{matrix} \longrightarrow \begin{matrix}+++ \\ +-+ \\ ++-\end{matrix}$	$4J$	$exp(-4\beta J)$
$\begin{matrix}+++ \\ ++- \\ +-+\end{matrix} \longrightarrow \begin{matrix}+++ \\ +-- \\ +-+\end{matrix}$	0	accept
$\begin{matrix}+++ \\ -+- \\ +-+\end{matrix} \longrightarrow \begin{matrix}+++ \\ --- \\ +-+\end{matrix}$	$-4J$	accept
$\begin{matrix}+-+ \\ -+- \\ +-+\end{matrix} \longrightarrow \begin{matrix}+-+ \\ --- \\ +-+\end{matrix}$	$-8J$	accept

4 Implementation and Performance

Our implementation of the Ising Model based on the Metropolis Algorithm follows precisely the instructions presented in the previous chapter. In the beginning an object depending on the initial conditions has to be created, according to the scheme (1D, 1D with external field or 2D) you would like to simulate. The relationships and heritages, as well as all member functions, are shown in Figure 2.

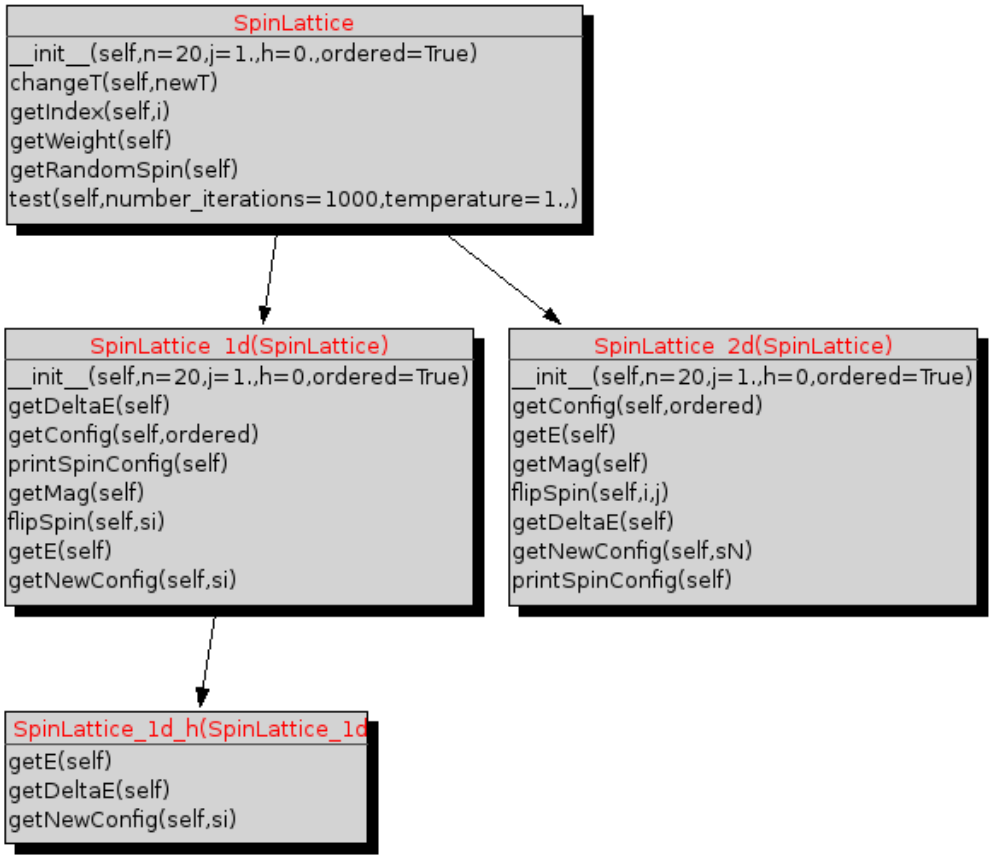


Figure 2: UML for SpinLattice.py

While creating one of these objects, simultaneously the initial spin lattice is formed by calling the $getConfig()$ member function. At that moment the temperature can be defined by the $changeT()$ member function. The next step is to select randomly a spin, check the cases according to the Metropolis rules and finally flip it or not. This functionality is provided by the $getNewConfig()$ method. We pay particular attention on not evaluating

the weights and the total energy inside the metropolis loop. For that reason we compute them once at each temperature step and store them in an array, the $getWeights()$ and $getDeltaE()$ functions are automatically called by the $changeT()$ method. Since the comparison of tho boolean variables is one of the cheapest operation a computer can perform, we implement the spins as boolean variables (True stands for up and False stands for down). As the system needs a certain 'time' to reach the thermal equilibrium, we start measuring the observables after several spin flips i.e. several $getNewConfig()$ calls. After the equilibrium is reached we perform several measurements and take the mean afterwards. For convenience we introduce a new parameter named *sweep*, that corresponds to the number of $getNewConfig()$ calls between two successive measurements. For instance for a 100×100 lattice we need at least a $sweep \geq 100 \times 100$. In order to avoid taking non-equilibrium measurements into account and obtaining thereby falsified results, the series of measurements starts after ten sweeps. To speed up the simulation we used Numpy Arrays, that are known for their faster handling compared to the common Python arrays. Additionally instead of generating another costly random number in the two dimensional case, we recycle the already generated random number by an divmod operation.

5 Results

5.1 One dimension

From the theory we know that in one dimension the Ising model shows no phase transition for finite temperatures i.e. magnetization is zero for all temperatures except $T = 0$. The following Plots (Figures 3,4) were generated with a lattice containing 1000 spins. For temperatures $T \geq 0.5$ our simulation results confirm this behavior Due to finite size effects our simulation wrongly predicts a finite magnetization for small temperatures. To obtain a results closer to the exact solution the spin lattice size should be increased. The $-\tanh(\frac{1}{T})$ dependence of the energy as a function of the temperature could be confirmed quite well. The heat capacity shows as well the theoretically predicted characteristics qualitatively. The susceptibility diverges for $T \to 0$. For a better understanding or visualization Figure 5 shows the dynamics of the lattice configuration during the simulation As one might expect for high temperatures the system remains total disordered, whereas for small temperatures the system clusters.

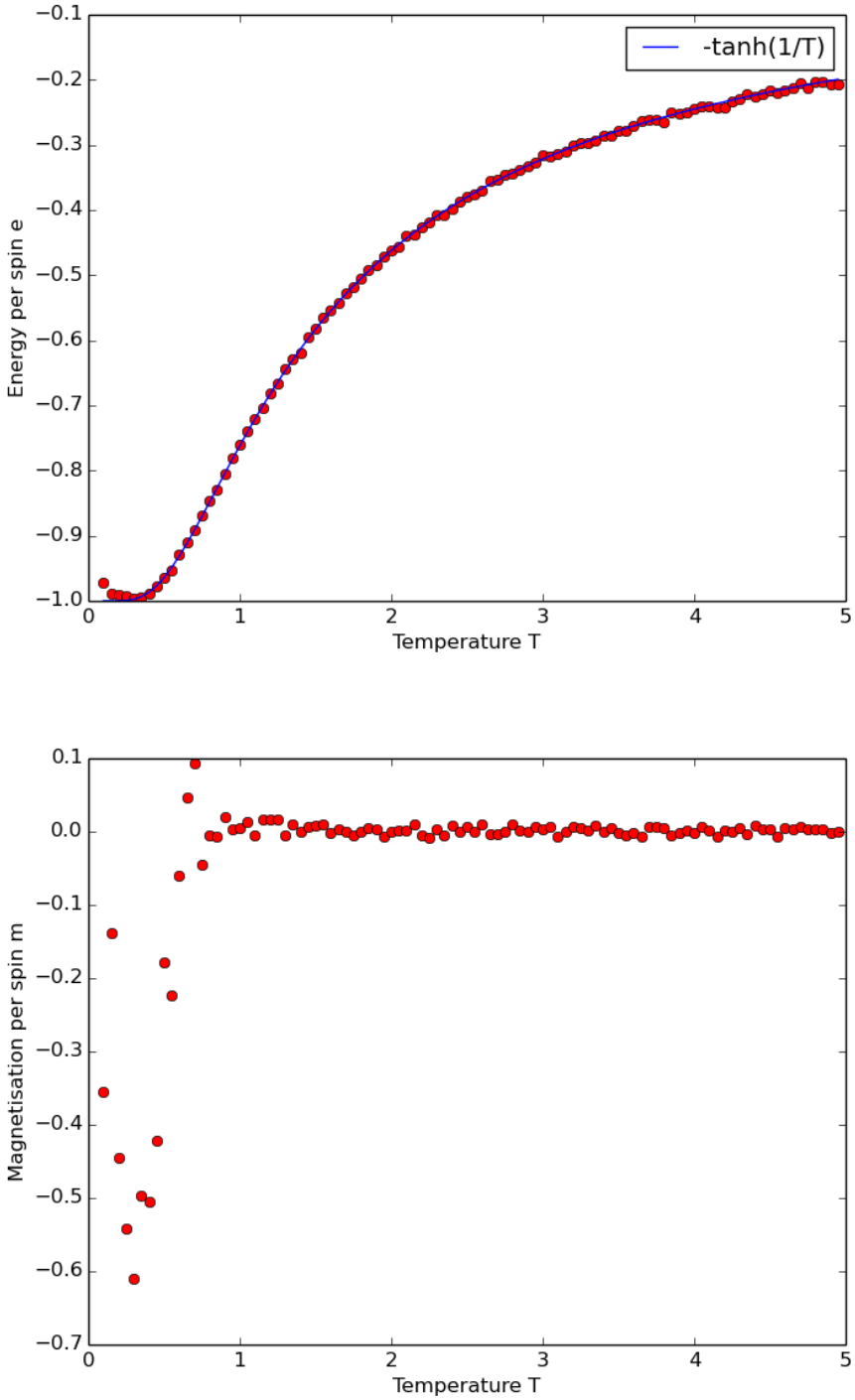


Figure 3: Energy and magnetization for 1D without an external field (j=1, sweep: 1000, number of sweeps: 100)

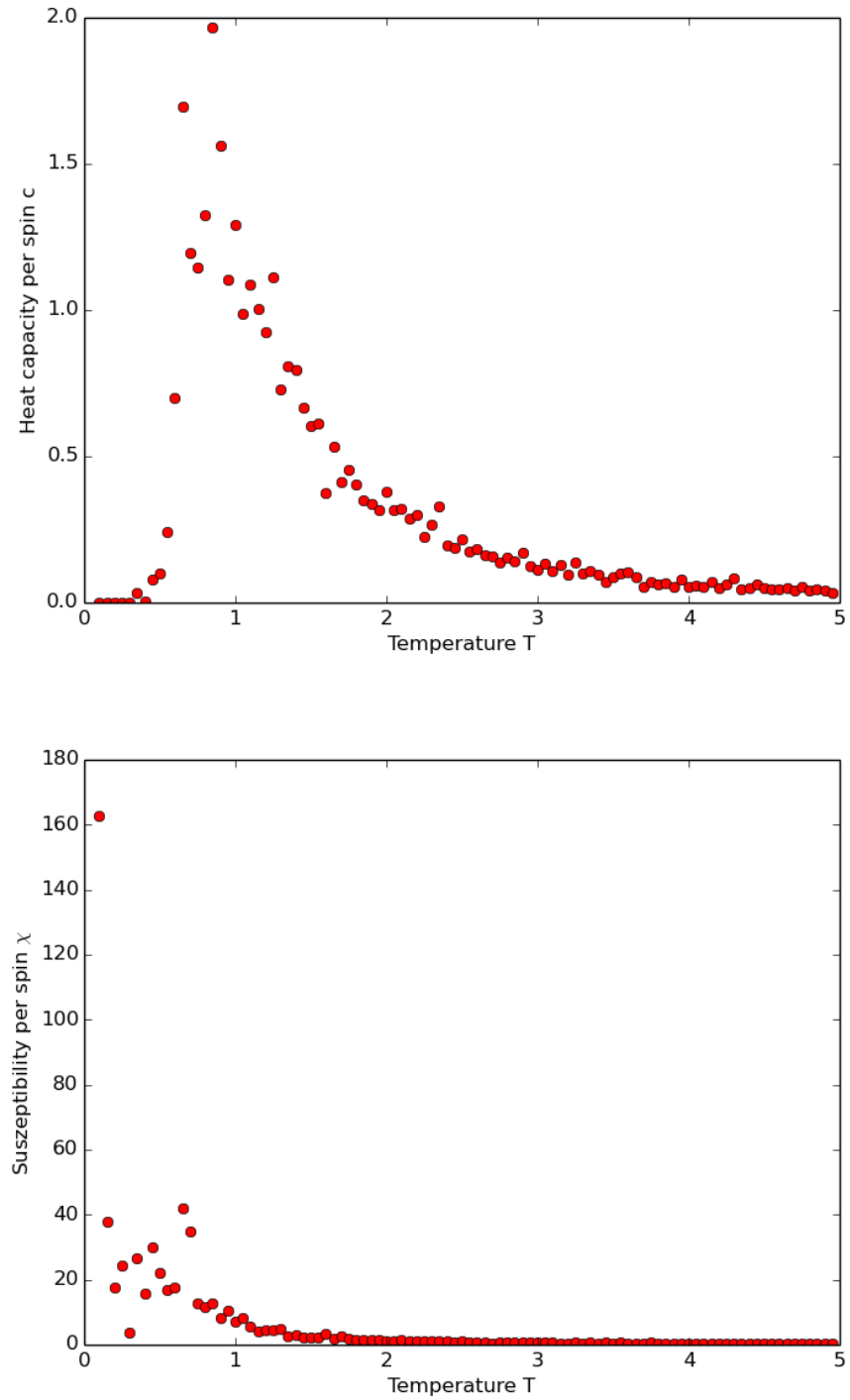


Figure 4: Heat capacity and susceptibility for 1D without an external field (j=1, sweep: 1000, number of sweeps: 100)

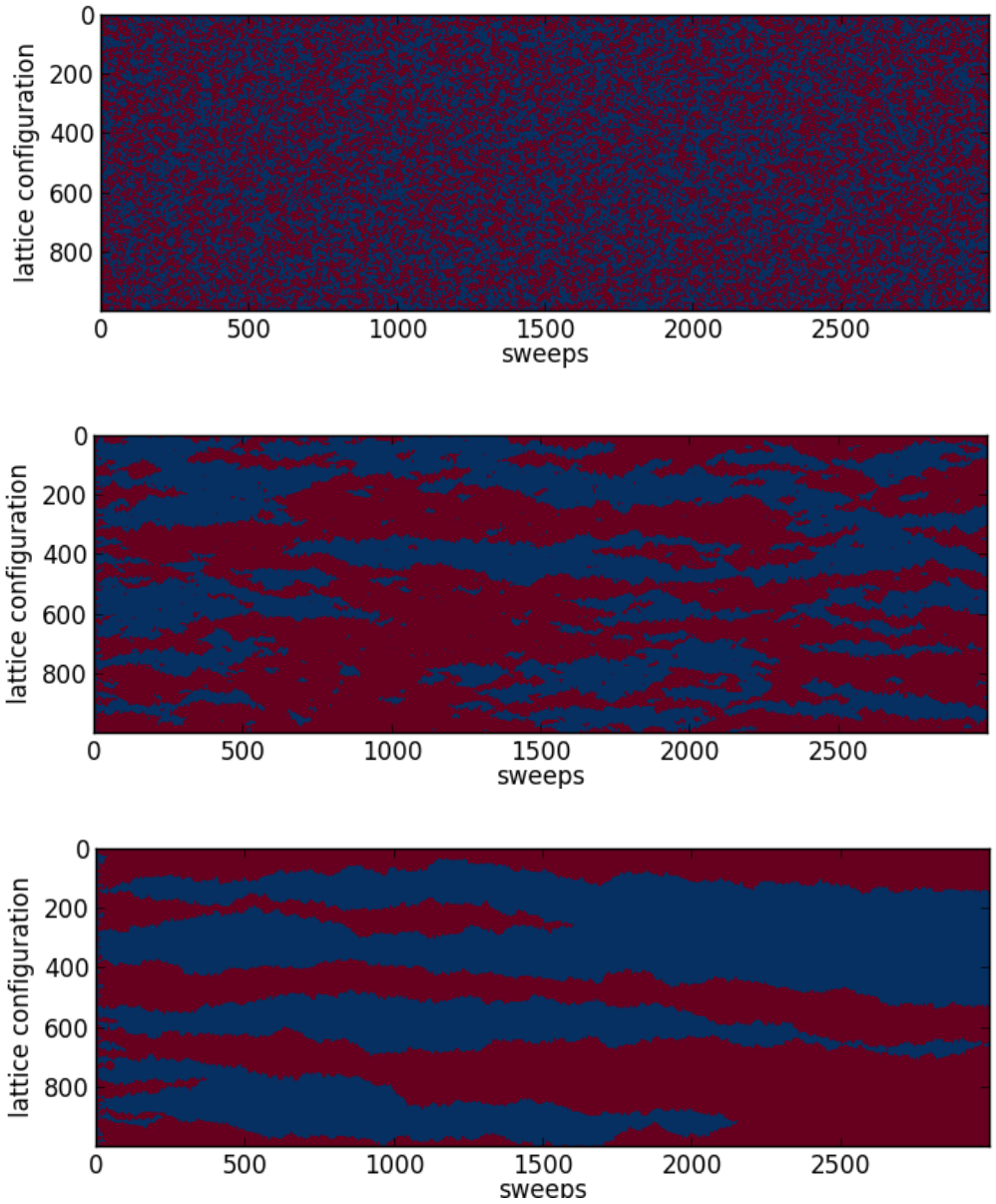


Figure 5: Trajectories for the 1D Ising model without an external field for different temperatures: T=1, 0.5, 0.2 and j=1

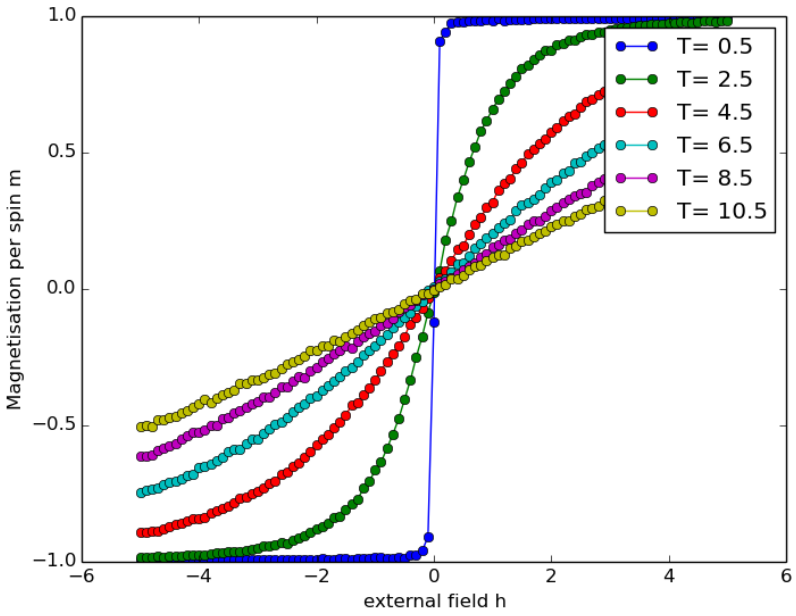


Figure 6: 1D Ising model: magnetization as a function of h for different temperatures

5.2 One dimension with external magnetic field

The magnetization as a function of the external field should have an sigmoidal shape. From the exact solution we know that for small temperatures the slop for $\langle m(h=0)\rangle$ increases. For higher temperatures thermal fluctuations work against parallelization of the spins by the external magnetic field. For that reason the system reacts slower on changes of the external field and the linear dependence of the energy on h is not longer valid. Our simulations (Figure 3, 7) results confirm these characteristics.

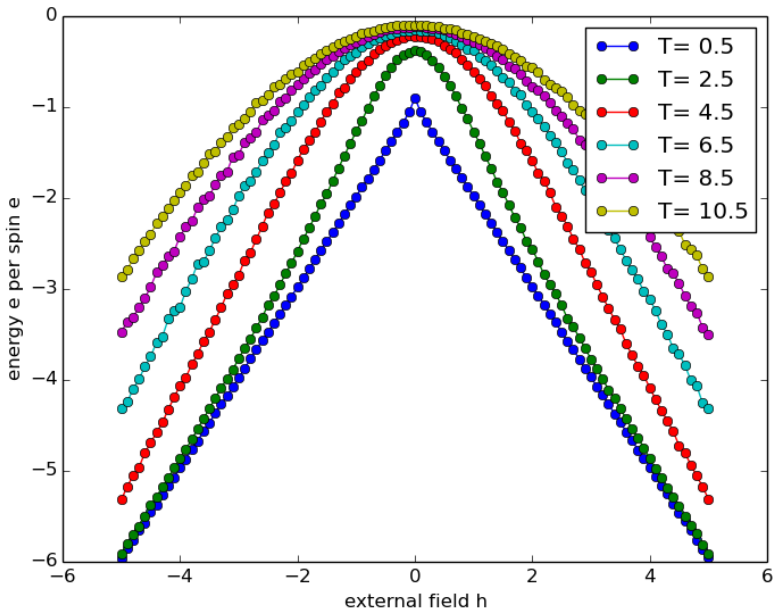


Figure 7: 1D Ising model: energy as a function of h for different temperatures

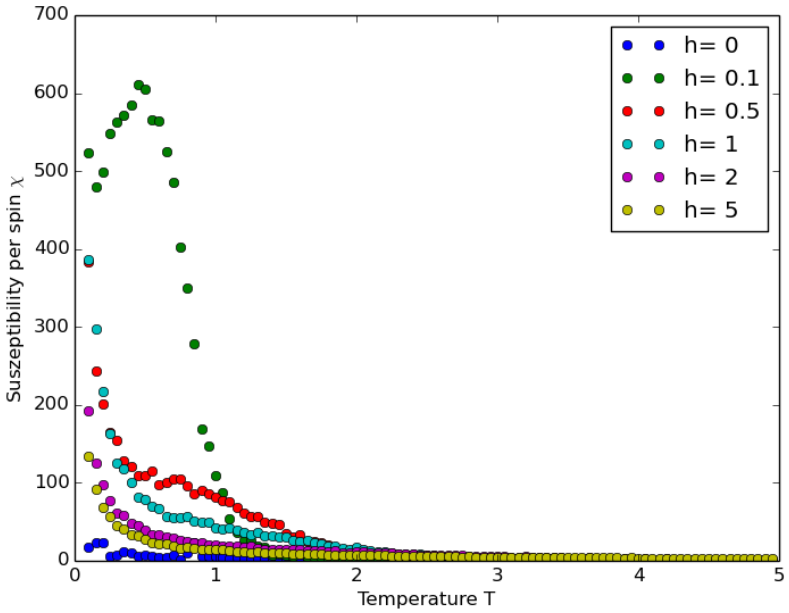
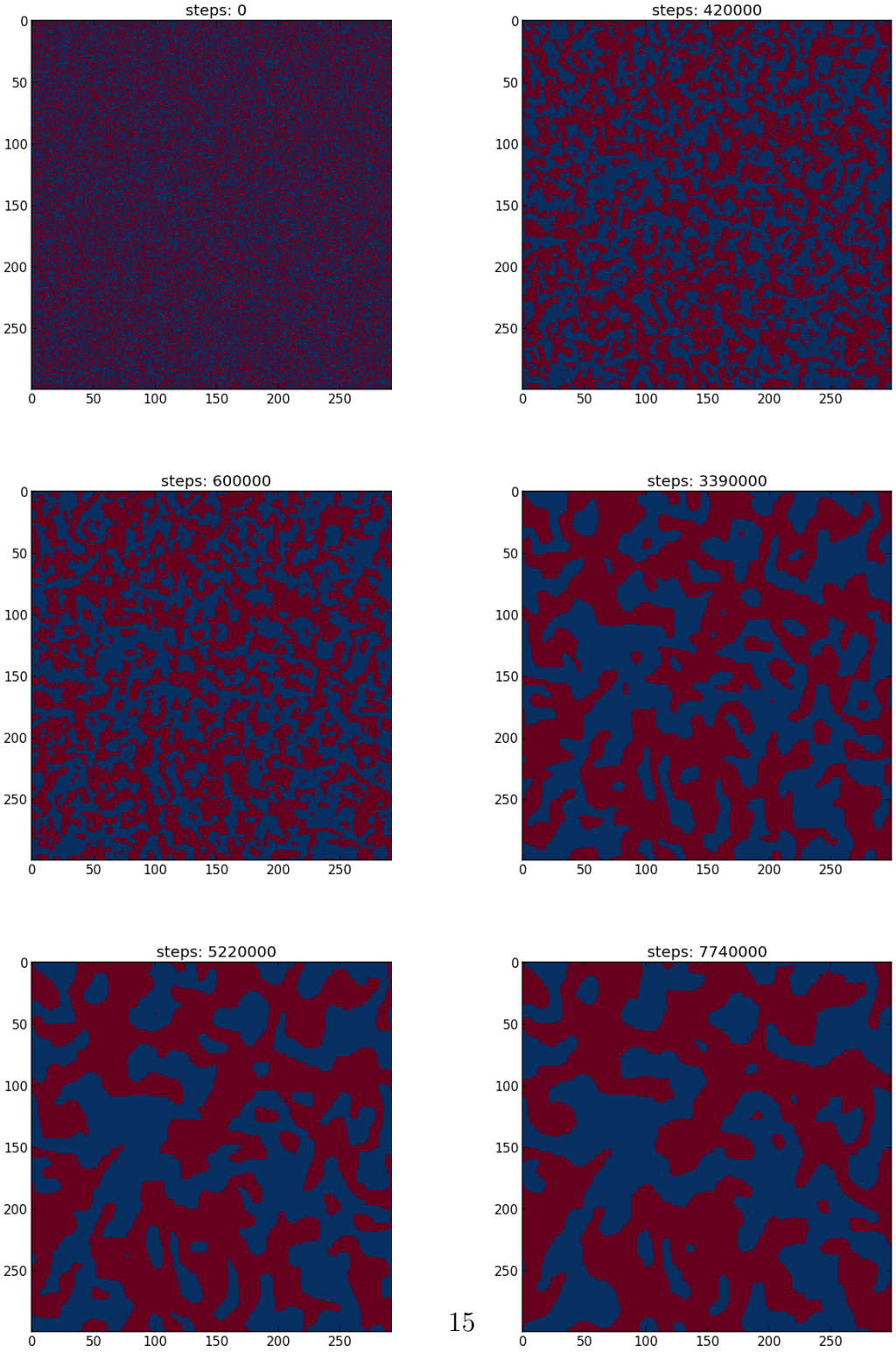
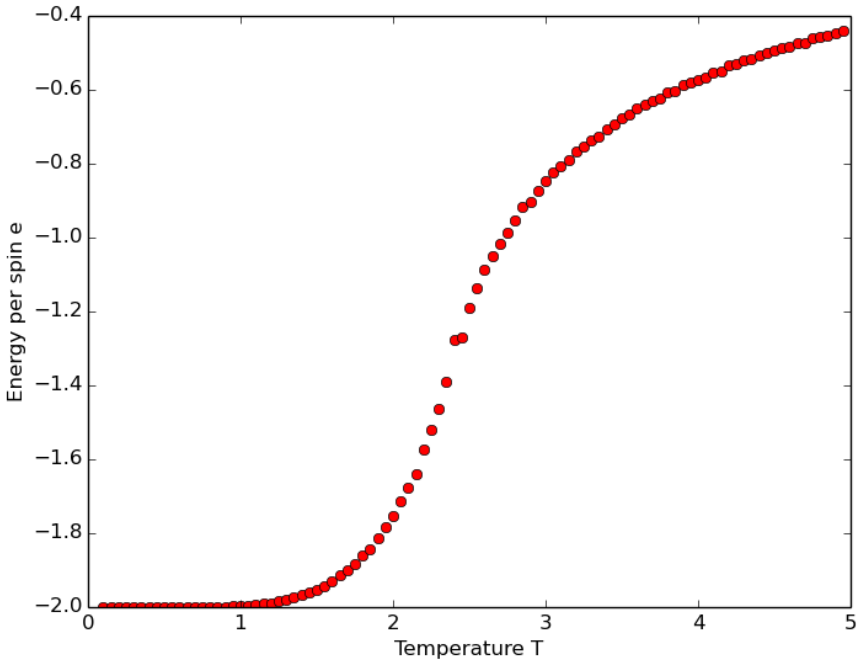


Figure 8: 1D Ising model: susceptibility as a function of h for different temperatures

5.3 Two dimensions

Table 4: Trajectories for the 2D Ising model without an external field with T= and j=1





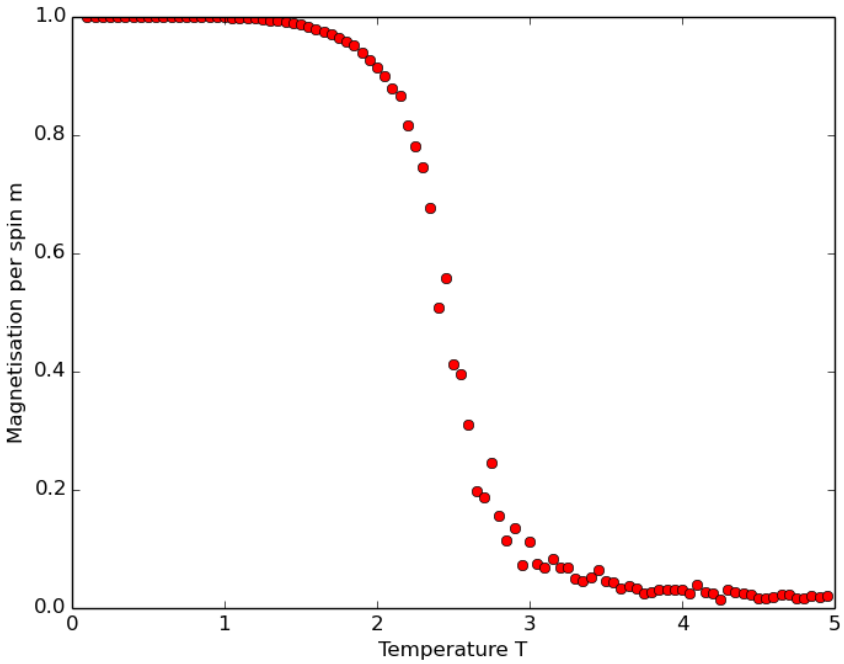
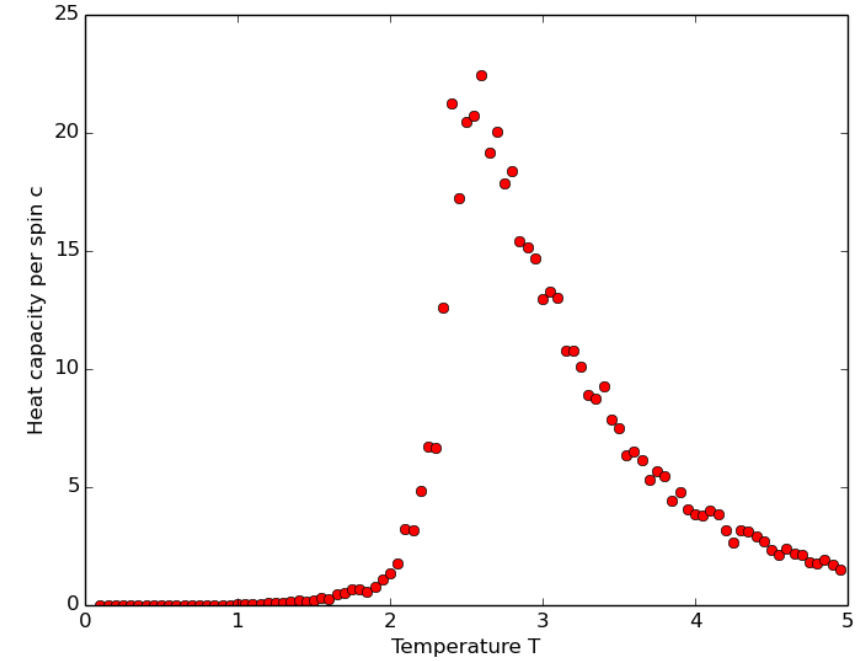


Table 5: Energy and magnetization for 2D without an external field and j=1



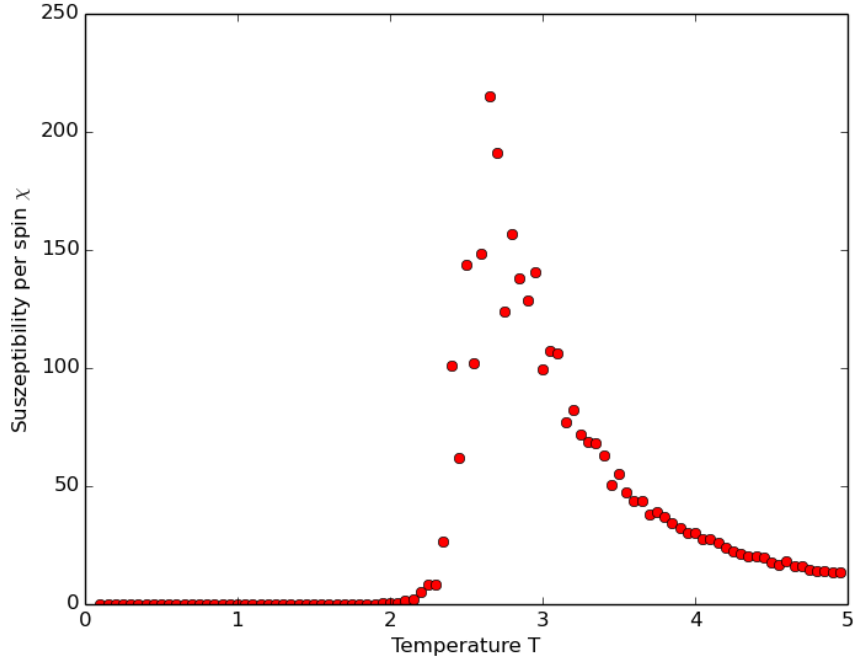


Table 6: Heat capacity and susceptibility for 2D without an external field and j=1

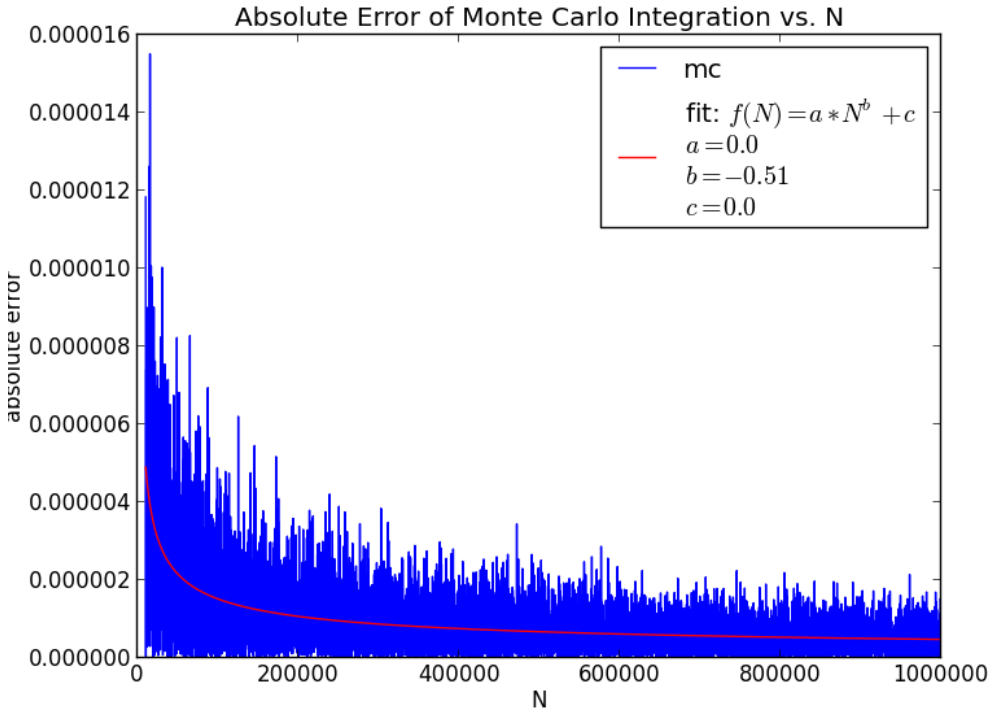
References

[1] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. The Journal of Chemical Physics, 1953.

6 Appendix

Monte Carlo Plots

Figure 9: $f(x) = x^2$, volume: $[-1, 1]$, ratio: 1.043



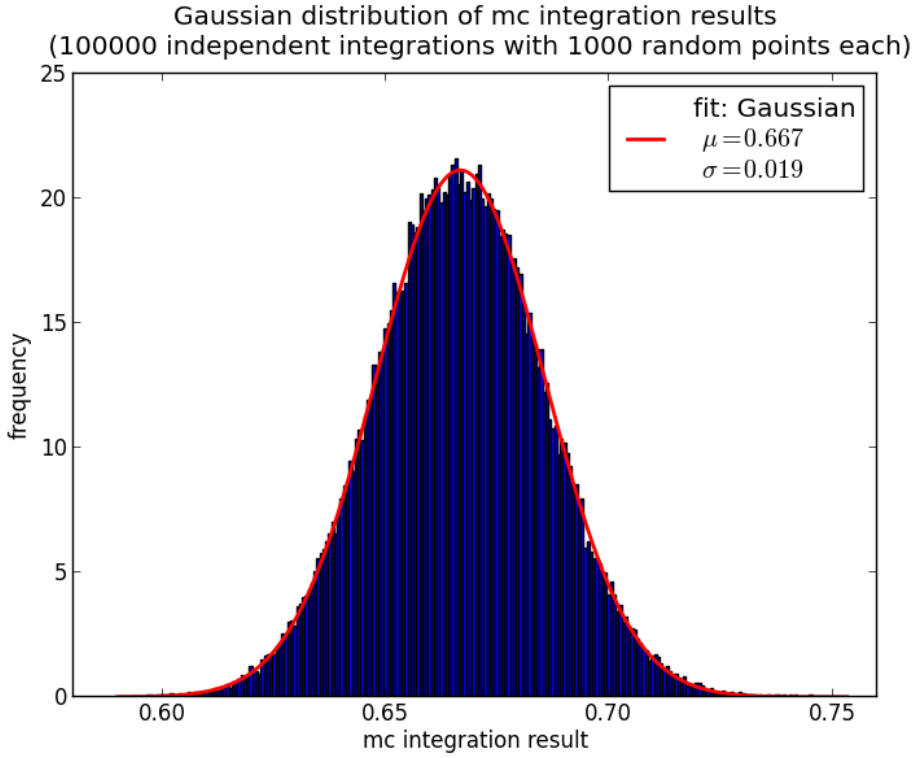
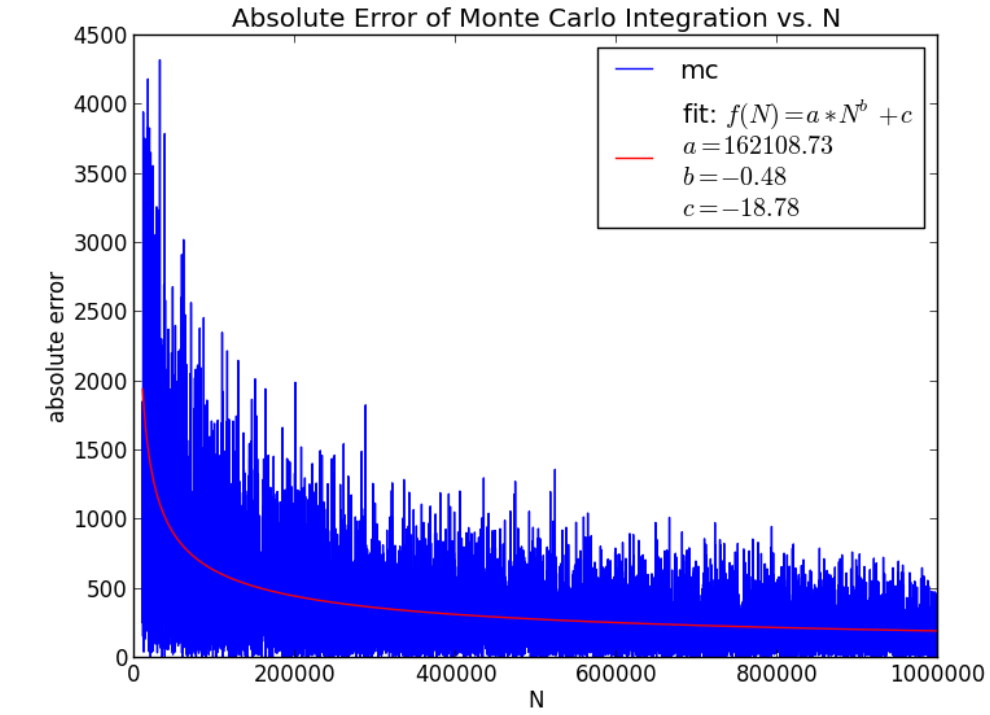


Figure 10: $f(x) = x^5$, volume: $[-1, 1]$, ratio: 0.951, (the fit function failed to find the correct fit paramters for the Gaussian)



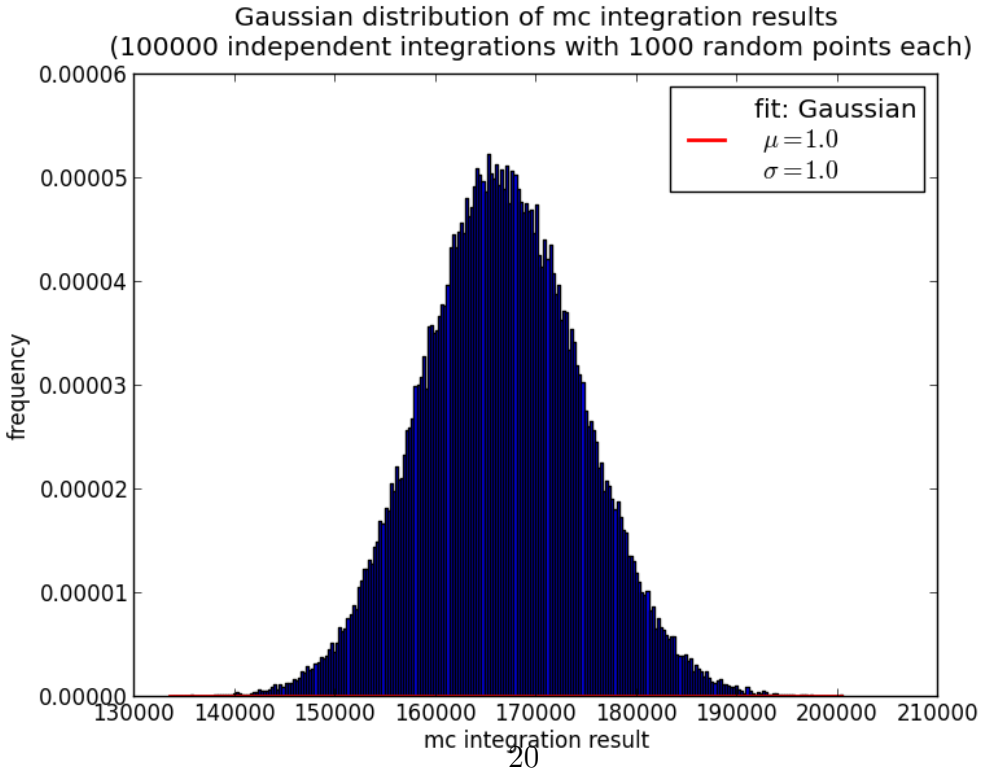
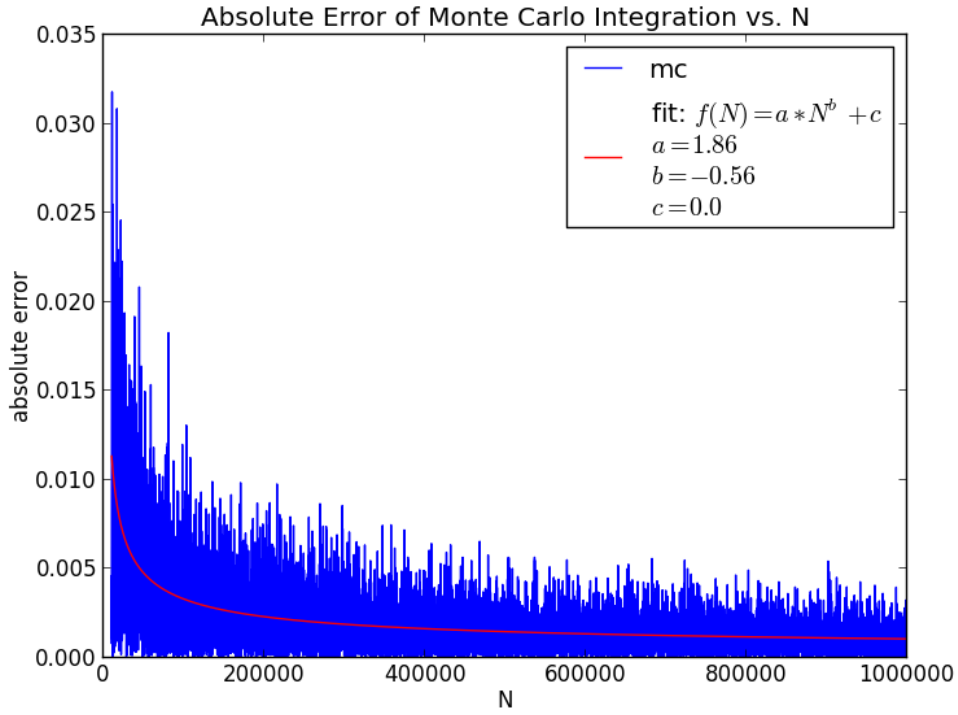


Figure 11: $f(x) = exp(x)$, volume: $[-1, 1]$, ratio: 0.932



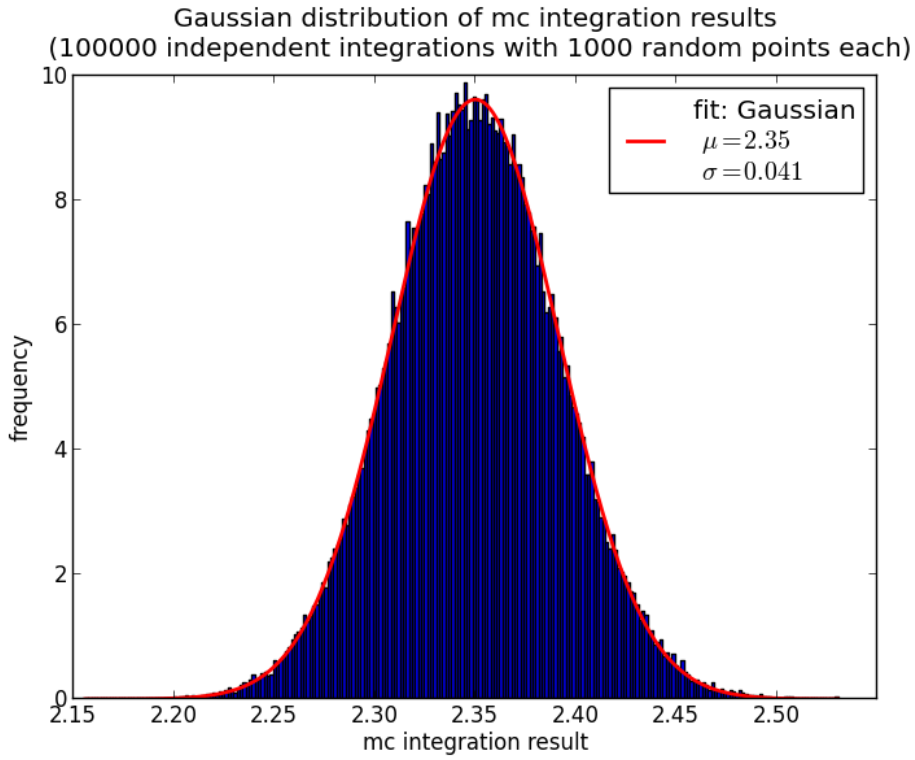
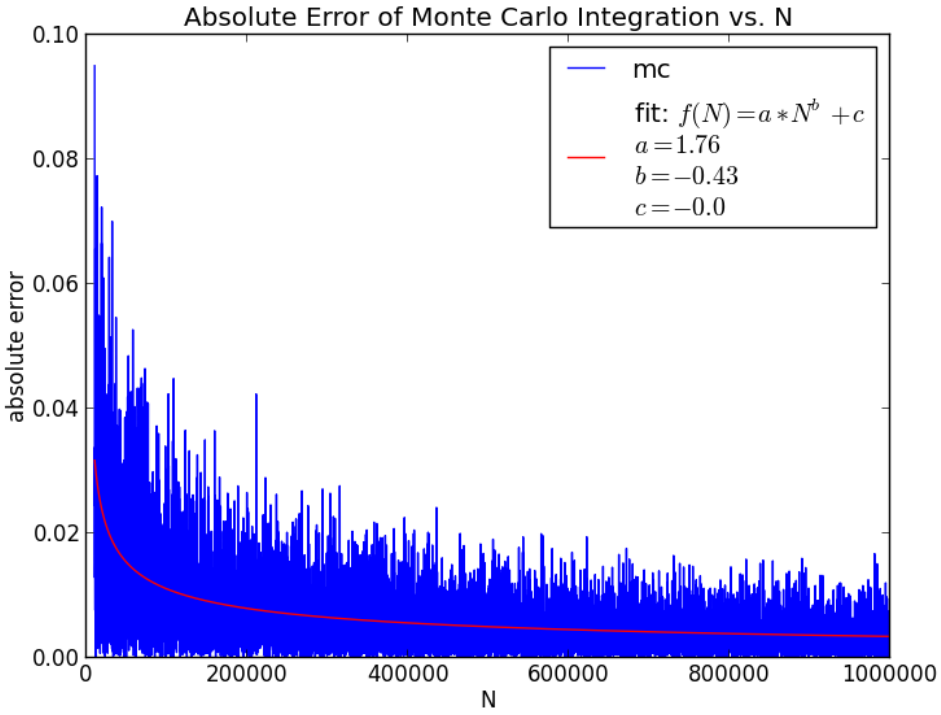
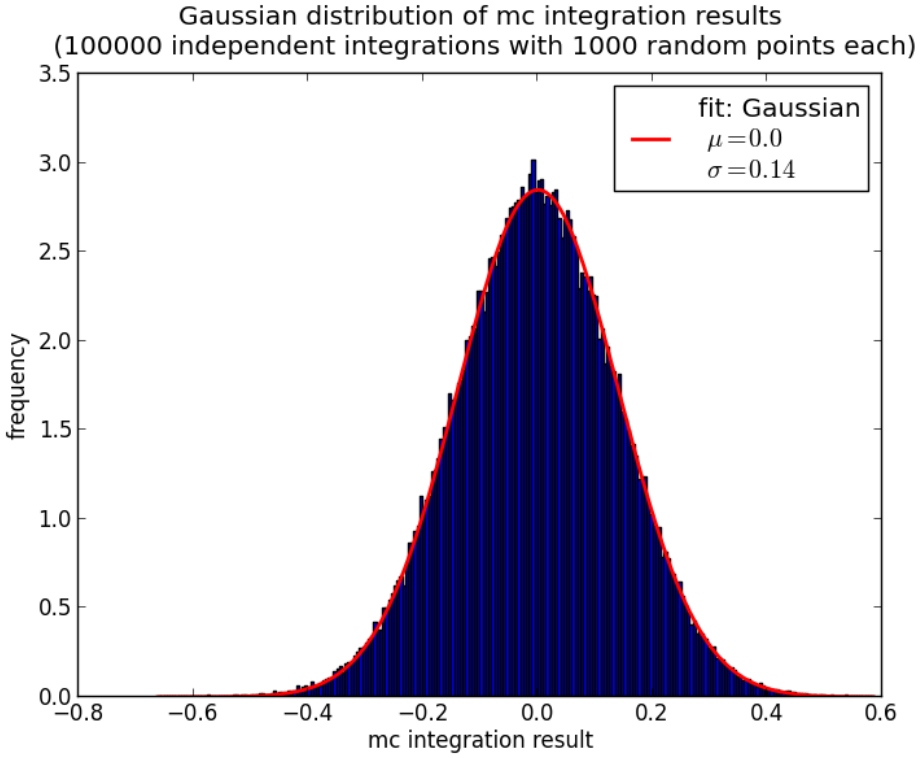


Figure 12: $f(x) = sin(x)$, volume: $[0, 2\pi]$, ratio: 1.085





Code Documentation

Spin_Lattice.py

```
Help on module Spin_Lattice:

NAME
    Spin_Lattice

FILE
    /home/s/Sophia.Kronthaler/Bananentasche/Spin_Lattice.py

CLASSES
    SpinLattice
        SpinLattice_1d
            SpinLattice_1d_h
        SpinLattice_2d

    class SpinLattice
     |  Methods defined here:
     |
     |  __init__(self, n=20, j=1.0, h=0.0, ordered=True)
     |      member variable    |   description
     |      ==================|============================
     |      n                  |   number of spins
     |      j                  |   coupling strength
     |      ordered            |   initial configuration
     |      beta               |   1./Temperature
     |      hDirection         |  for h>= 0 all spins are up if ordered=True
     |      h                  |   absolute value of the external magnetic field
     |      config             |   container for the initial configuration (getConfig())
     |      E                  |   container for the total energy
     |
     |  changeT(self, newT)
     |      to adjust the member variable beta according to the newT: beta = 1./newT
     |      in addition computes the new (temperature dependent) weights
     |      returns None
     |
     |  getIndex(self, i)
     |      simplifies life by returning the correct index with regard to periodic
     |      boundary conditions (python can deal with negative indices,
     |      so we do not have to implement the -1 case)
     |
     |  getRandomSpin(self)
     |      returns coordinate of a randomly chosen spin: int(si)
     |
     |  getWeight(self)
     |      computes the Boltzmann weight factor needed in the Metropolis Algorithm
     |      changes the member variable weight
     |      returns None
     |
     |  test(self, number_iterations=1000, temperature=1.0)
     |      test routine
     |      number of iterations: number of spin flips
     |      prints out several control parameters

    class SpinLattice_1d(SpinLattice)
     |  for the one dimensional case(h=0)
```

```
|
|  Methods defined here:
|
|  __init__(self, n=20, j=1.0, h=0, ordered=True)
|      member variable |    description
|      ================|============================================================
|      hStrong         |  field is strong if h > j (to distinguish
|                      |  between "strong" and "weak" fields when
|                      |  checking in the Metropolis Algorithm weather
|                      |  the new state is accepted directly or not
|      weight          |  container for Boltzmann weight factor, saved in the beginnig
|                      |  (getWeight())
|      deltaE          |  container for energy difference
|
|  flipSpin(self, si)
|      performs the spin flip of the spin si
|      returns None
|
|  getConfig(self, ordered)
|      returns the initial configuration: array(bool)
|      if ordered=True: all spins up
|      if ordered=False: random spin config
|
|  getDeltaE(self)
|
|  getE(self)
|      returns the total energy of the initial configuration
|      parallel: -j
|      anti-parallel: +j
|
|  getMag(self)
|      returns the magnetization of the current spin configuration:
|      int(Mag)
|
|  getNewConfig(self, si)
|      this function computes the energy difference due to the prospective flip of
|      the spin si and flips the randomly chosen spin with if the metropolis algorithm
|      condition is full filled
|
|  printSpinConfig(self)
|      prints the curren spin configuration
|      up-spins are represented by "*", down-spins by "|"
|
|  ----------------------------------------------------------------------
|  Methods inherited from SpinLattice:
|
|  changeT(self, newT)
|      to adjust the member variable beta according to the newT: beta = 1./newT
|      in addition computes the new (temperature dependent) weights
|      returns None
|
|  getIndex(self, i)
|      simplifies life by returning the correct index with regard to periodic
|      boundary conditions (python can deal with negative indices,
|      so we do not have to implement the -1 case)
|
|  getRandomSpin(self)
|      returns coordinate of a randomly chosen spin: int(si)
|
```

```
 |  getWeight(self)
 |      computes the Boltzmann weight factor needed in the Metropolis Algorithm
 |      changes the member variable weight
 |      returns None
 |
 |  test(self, number_iterations=1000, temperature=1.0)
 |      test routine
 |      number of iterations: number of spin flips
 |      prints out several control parameters

class SpinLattice_1d_h(SpinLattice_1d)
 |  for the one dimensional case with an external magnetic field
 |
 |  Method resolution order:
 |      SpinLattice_1d_h
 |      SpinLattice_1d
 |      SpinLattice
 |
 |  Methods defined here:
 |
 |  getDeltaE(self)
 |      returns the energy difference of flipping one spin
 |      deltaE[0]:   4*j + 2*h
 |      deltaE[1]:   4*j - 2*h
 |      deltaE[2]: - 4*j - 2*h
 |      deltaE[3]: - 4*j + 2*h
 |      deltaE[4]:         2*h
 |      deltaE[5]:       - 2*h
 |
 |  getE(self)
 |      calculates the total energy for a given temperature of the
 |      initial spin configuration
 |      is called in changeT()
 |      only compute the initial energy once in the beginning, during
 |      the simulation we only add or distract small amounts of energy (deltaE)
 |
 |  getNewConfig(self, si)
 |      returns the new configuration due to the Metropolis algorithm
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from SpinLattice_1d:
 |
 |  __init__(self, n=20, j=1.0, h=0, ordered=True)
 |      member variable |    description
 |      ================|============================================================
 |      hStrong         |  field is strong if h > j (to distinguish
 |                      |  between "strong" and "weak" fields when
 |                      |  checking in the Metropolis Algorithm weather
 |                      |  the new state is accepted directly or not
 |      weight          |  container for Boltzmann weight factor, saved in the beginnig
 |                      |  (getWeight())
 |      deltaE          |  container for energy difference
 |
 |  flipSpin(self, si)
 |      performs the spin flip of the spin si
 |      returns None
 |
 |  getConfig(self, ordered)
 |      returns the initial configuration: array(bool)
```

```
 |      if ordered=True: all spins up
 |      if ordered=False: random spin config
 |
 |  getMag(self)
 |      returns the magnetization of the current spin configuration:
 |      int(Mag)
 |
 |  printSpinConfig(self)
 |      prints the curren spin configuration
 |      up-spins are represented by "*", down-spins by "|"
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from SpinLattice:
 |
 |  changeT(self, newT)
 |      to adjust the member variable beta according to the newT: beta = 1./newT
 |      in addition computes the new (temperature dependent) weights
 |      returns None
 |
 |  getIndex(self, i)
 |      simplifies life by returning the correct index with regard to periodic
 |      boundary conditions (python can deal with negative indices,
 |      so we do not have to implement the -1 case)
 |
 |  getRandomSpin(self)
 |      returns coordinate of a randomly chosen spin: int(si)
 |
 |  getWeight(self)
 |      computes the Boltzmann weight factor needed in the Metropolis Algorithm
 |      changes the member variable weight
 |      returns None
 |
 |  test(self, number_iterations=1000, temperature=1.0)
 |      test routine
 |      number of iterations: number of spin flips
 |      prints out several control parameters

class SpinLattice_2d(SpinLattice)
 |  Methods defined here:
 |
 |  __init__(self, n=20, j=1.0, h=0, ordered=True)
 |
 |  flipSpin(self, i, j)
 |      performs the spin flip of the spin si
 |      returns None
 |
 |  getConfig(self, ordered)
 |      returns the initial configuration: array(bool)
 |      if ordered=True: all spins up
 |      if ordered=False: random spin config
 |
 |  getDeltaE(self)
 |      returns the energy difference of flipping one spin
 |      DeltaE[0]=   8*j
 |      DeltaE[1]=   4*j
 |      DeltaE[2]=   0
 |      DeltaE[3]= - 4*j
 |      DeltaE[4]= - 8*j
 |
```

```
 |  getE(self)
 |      returns the total energy of the initial configuration
 |      parallel: -j
 |      anti-parallel: +j
 |
 |  getMag(self)
 |      returns the magnetization of the current spin configuration:
 |      int(Mag)
 |
 |  getNewConfig(self, sN)
 |      returns the new configuration due to the Metropolis algorithm
 |
 |  printSpinConfig(self)
 |      prints the curren spin configuration
 |      up-spins are represented by "*", down-spins by "o"
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from SpinLattice:
 |
 |  changeT(self, newT)
 |      to adjust the member variable beta according to the newT: beta = 1./newT
 |      in addition computes the new (temperature dependent) weights
 |      returns None
 |
 |  getIndex(self, i)
 |      simplifies life by returning the correct index with regard to periodic
 |      boundary conditions (python can deal with negative indices,
 |      so we do not have to implement the -1 case)
 |
 |  getRandomSpin(self)
 |      returns coordinate of a randomly chosen spin: int(si)
 |
 |  getWeight(self)
 |      computes the Boltzmann weight factor needed in the Metropolis Algorithm
 |      changes the member variable weight
 |      returns None
 |
 |  test(self, number_iterations=1000, temperature=1.0)
 |      test routine
 |      number of iterations: number of spin flips
 |      prints out several control parameters

DATA
    __author__ = 'Sophia_Kronthaler_and_Tobias_Goeppel'

AUTHOR
    Sophia_Kronthaler_and_Tobias_Goeppel
```