Solutions - Exercise sheet 3

The one-dimensional Ising model

PUE Advanced Computational Physics

University of Vienna - Faculty of Physics

The Ising model

In its simplest form, the Ising model is defined by the energy of state $\nu = \{s_1, \ldots, s_N\}$,

$$E_\nu = -J \sum_{i,j}{}' s_i s_j - H \sum_i s_i, \tag{1}$$

where $J > 0$ is the coupling constant, H is the external field, $\sum_{i,j}'$ indicates a sum over all nearest neighbor pairs, and the s_i are the spins that can take on values of $+1$ and -1. Note that for simplicity, we have set $\mu = 1$ for the magnetic moment of a spin.

8 Implementing an MC simulation of the 1d Ising model

In the following you will create a Metropolis Monte Carlo simulation of the one-dimensional Ising model. The file `mc_ising_1d_incomplete.zip` contains the scaffolding for this simulation, which you will complete in the following steps. Alternatively, feel free to implement everything from scratch in your language of choice.

In the Python example, the simulation is implemented in the class `IsingModelMC_1D`. In the `__init__` method you can see the variables that are used during the simulation (remember: the *self* variable is used to access members of the instance itself):

- **N** - number of spins in the system
- **pbc** - True if periodic boundary conditions are used
- **h_over_j** - the ratio H/J
- **T** - the temperature
- **beta** - the inverse temperature $\beta = 1/k_\mathrm{B}T$ with $k_\mathrm{B} = 1$
- **config** - the configuration as an integer array of size N

We will start building the necessary routines from the bottom up. Finish the implementation of the following methods:

a. **get_neighbors(self,i)** - return the neighbors of spin *i* with or without periodic boundary conditions depending on the value of **self.pbc**.

b. **get_E(self)**, **get_dE_single(self,i)** - these methods calculate the total potential energy and the change in total energy if spin *i* is flipped, respectively. Make sure that these two methods are consistent with each other before you proceed.

c. **acceptance(self,dE)** - implement the Metropolis acceptance criterion based on the difference in potential energy, **dE**, and the temperature **T**.

d. **do_move(self)** - implement a single Metropolis Monte-Carlo move using the methods you implemented above. Which steps are necessary? If you want to accept a move, use the **accept_move** method, otherwise just leave the function.

Section 3 of the notebook contains code that initializes a simulation instance. You can use the **system** variable to test various parts of the code by calling methods directly. It is highly recommended that you test each method you implement separately before moving on to the next task.

e. Test your implementation by running a simulation. With the parameters given, you can expect an acceptance ratio of roughly 25%.

[ ]:
```
def get_neighbors(self,i):
    "Return the neighbors of spin i"
    if self.pbc:
        neighbors = [(i-1)%self.N, (i+1)%self.N]
    else:
        if i == 0:
            neighbors = [1, ]
        elif i == (self.N - 1):
            neighbors = [N-2, ]
        else:
            neighbors = [i-1, i+1]
    return neighbors


def get_E(self):
    "Return the total current total energy of the system"
    sum_E = 0.0
    for i in range(self.N):
        sum_E += self.get_E_single(i)
    return sum_E


def get_dE_single(self,i):
    "Return the change in energy of particle i is flipped"
    nbs = self.get_neighbors(i)
    sum_E = 0.0
    me = self.config[i]
    for j in nbs:
        sum_E += 2*me*self.config[j]
    sum_E += 2*self.h_over_j*me
    return sum_E


def acceptance(self,dE,debug_level=0):
    "Return true if Metropolis MC acceptance criterion is accepted - false␣
↪otherwise"
    if dE < 0:
        if debug_level >= 3:
            print("* Acceptance test: dE < 0 -> ACCEPTED",end='')
        return True
    else:
        p = exp(-self.beta*dE)
        _random = random.random()
        if debug_level >= 3:
            print("* Acceptance test: p = % 5.3f, random number = % 5.3f␣
↪->"%(p,_random),end='')
        if _random < p:
            if debug_level >= 3:
                print("ACCEPTED")
            return True
        else:
            if debug_level >= 3:
                print("REJECTED")
            return False

def do_move(self,debug_level=0):
        "Perform a single Metropolis MC move including an acceptance test"
        i = random.randint(self.N)

        if debug_level >= 2:
            print("* Chose spin %i"%i)

        dE = self.get_dE_single(i)
        if debug_level >= 2:
            print("* me = %i"%self.config[i],end="")
            for j,_nb in enumerate(self.get_neighbors(i)):
                print(", NB %i = %i"%(j,self.config[_nb]),end="")
            print("")
            print("* dE = %.2f"%dE)

        self.n_attempted += 1
        if self.acceptance(dE,debug_level=debug_level):
            self.accept_move(i,dE,debug_level=debug_level)
        else:
            pass
```

9 Calculating the magnetization

The partition function of the 1d Ising model with N spins is given by

$$Z_N = \lambda_+^N + \lambda_-^N = \lambda_+^N \left[1 + \left(\frac{\lambda_-}{\lambda_+} \right)^N \right] \tag{2}$$

with

$$\lambda_\pm = e^{\beta J} \cosh(\beta H) \pm \sqrt{e^{2\beta J} \sinh^2(\beta H) + e^{-2\beta J}}. \tag{3}$$

a. Show that in the limit $N \to \infty$ the magnetization per spin $m = \sum_i s_i / N$ is given by

$$m = \frac{e^{\beta J} \sinh(\beta H)}{\sqrt{e^{2\beta J} \sinh^2(\beta H) + e^{-2\beta J}}}. \tag{4}$$

$$E_v = -J{\sum_{i,j}}' s_i s_j - H \sum_i s_i \tag{1}$$

The partition function of the 1D-Ising model is given by:

$$Z_N = \lambda_+^N \left[1 + \left(\frac{\lambda_-}{\lambda_+} \right)^N \right] \tag{2}$$

where

$$\lambda_+ = e^{\beta J} \cosh(\beta H) + \sqrt{e^{2\beta J} \sinh^2(\beta H) + e^{-2\beta J}} \tag{3}$$

$$\lambda_- = e^{\beta J} \cosh(\beta H) - \sqrt{e^{2\beta J} \sinh^2(\beta H) + e^{-2\beta J}} \tag{4}$$

In general, the magnetization is given by:

$$m = \frac{1}{N} \frac{1}{\beta} \frac{\partial}{\partial H} \ln Z \tag{5}$$

$$= \frac{1}{N} \frac{1}{\beta} \frac{1}{Z_N} \sum_v e^{-\beta E_v} \sum_i \beta s_i \tag{6}$$

$$= \frac{1}{N} \frac{1}{Z_N} \sum_v e^{-\beta E_v} \sum_i s_i \tag{7}$$

where $\frac{1}{N} \sum_i s_i$ is the magnetization per spin.
To calculate the magnetization in the thermodynamic limit, we first note that λ_+ is always grater than λ_-:

$$\lambda_+ > \lambda_- \tag{8}$$

That means for $N \to \infty$:

$$\lim_{N\to\infty} Z_N = \lim_{N\to\infty} \lambda_+^N \left[1 + \left(\frac{\lambda_-}{\lambda_+} \right)^N \right] = \lambda_+^N \tag{9}$$

Inserting the expression for Z_N into equation 5:

$$m = \frac{1}{\beta} \frac{1}{N} \frac{\partial}{\partial H} \ln \lambda_+^N \tag{10}$$

$$= \frac{1}{\beta} \frac{\partial}{\partial H} \ln \lambda_+ \tag{11}$$

$$= \frac{1}{\beta} \left[\frac{\beta e^{\beta J} \sinh(\beta H)}{\sqrt{e^{2\beta J} \sinh^2(\beta H) + e^{-2\beta J}}} \right] \tag{12}$$

$$= \frac{\beta e^{\beta J} \sinh(\beta H)}{\sqrt{e^{2\beta J} \sinh^2(\beta H) + e^{-2\beta J}}} \tag{13}$$

a. Use this analytical result to check your simulation. Discuss your findings.

Exercise 9b: magnetization (as a function of external field system size)

[16]:
```python
def get_pars(h,size):
    return dict(
        system_size = size,
        h_over_j = h,
        init_strategy = "random",
        boundary_conditions = "pbc",
        T=1.0,
        i_stats = 10,
        i_stats_out = 0
    )

hs = np.linspace(0.0, 1.4, 15)

# this will take a few minutes on your average machine
def get_m_of_h(size):
    ms = []
    m_timeseries = []
    for h in hs:
        print("** Running simulation at H = %g"%h)
        _pars = get_pars(h,size)

        system = IsingModelMC_1D(_pars)

        system.run(10000 // size, _pars)
        system.reset_statistics_variables()
        system.run(80000 // size, _pars)

        _avgs = system.get_averages()
        ms.append(_avgs['m'])
        m_timeseries.append(system.get_timeseries()['m'])
    return ms,m_timeseries

sizes = [2, 5, 10, 50, 100]
ms_of_h = []
m_timeseries_of_h = []
for size in sizes:
    print("----- SIZE % 4i -----"%size)
    avg_m, m_timeseries = get_m_of_h(size)
    ms_of_h.append(avg_m)
    m_timeseries_of_h.append(m_timeseries)
```

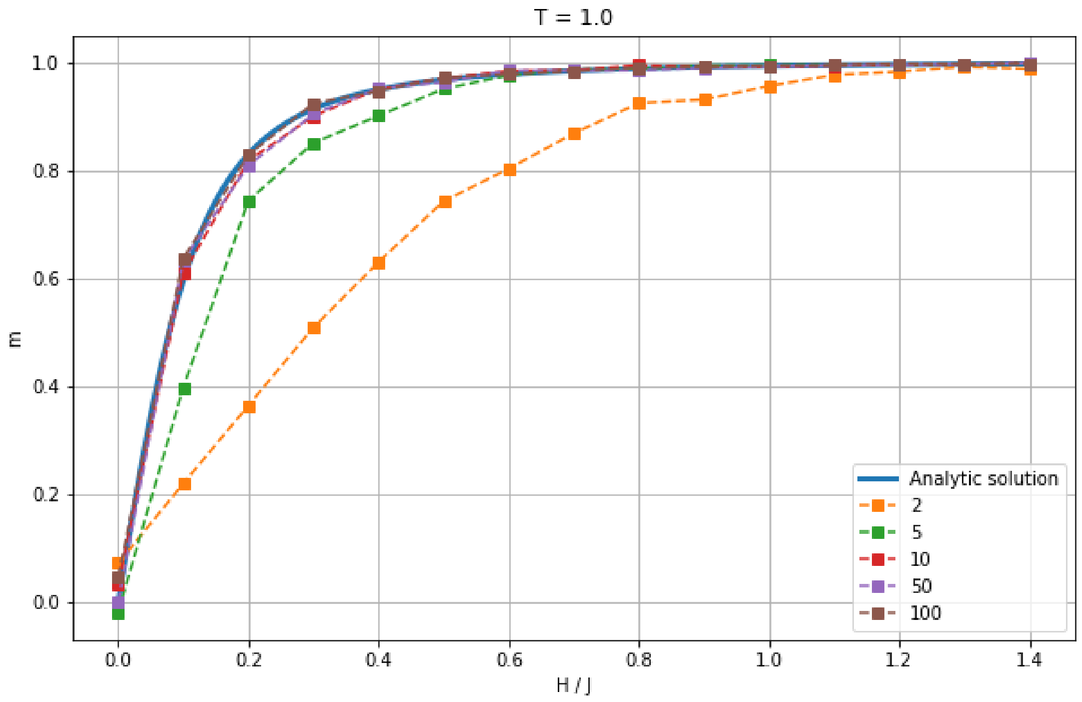
```
----- SIZE    2 -----
** Running simulation at H = 0
** Running simulation at H = 0.1
** Running simulation at H = 0.2
** Running simulation at H = 0.3
** Running simulation at H = 0.4
** Running simulation at H = 0.5
** Running simulation at H = 0.6
** Running simulation at H = 0.7
** Running simulation at H = 0.8
** Running simulation at H = 0.9
** Running simulation at H = 1
** Running simulation at H = 1.1
** Running simulation at H = 1.2
** Running simulation at H = 1.3
** Running simulation at H = 1.4
----- SIZE    5 -----
** Running simulation at H = 0
** Running simulation at H = 0.1
** Running simulation at H = 0.2
** Running simulation at H = 0.3
** Running simulation at H = 0.4
** Running simulation at H = 0.5
** Running simulation at H = 0.6
** Running simulation at H = 0.7
** Running simulation at H = 0.8
** Running simulation at H = 0.9
** Running simulation at H = 1
** Running simulation at H = 1.1
** Running simulation at H = 1.2
** Running simulation at H = 1.3
** Running simulation at H = 1.4
----- SIZE   10 -----
** Running simulation at H = 0
** Running simulation at H = 0.1
** Running simulation at H = 0.2
** Running simulation at H = 0.3
** Running simulation at H = 0.4
** Running simulation at H = 0.5
** Running simulation at H = 0.6
** Running simulation at H = 0.7
** Running simulation at H = 0.8
** Running simulation at H = 0.9
** Running simulation at H = 1
** Running simulation at H = 1.1
** Running simulation at H = 1.2
** Running simulation at H = 1.3
** Running simulation at H = 1.4
----- SIZE   50 -----
** Running simulation at H = 0
** Running simulation at H = 0.1
** Running simulation at H = 0.2
** Running simulation at H = 0.3
** Running simulation at H = 0.4
** Running simulation at H = 0.5
** Running simulation at H = 0.6
** Running simulation at H = 0.7
** Running simulation at H = 0.8
** Running simulation at H = 0.9
** Running simulation at H = 1
** Running simulation at H = 1.1
** Running simulation at H = 1.2
** Running simulation at H = 1.3
** Running simulation at H = 1.4
----- SIZE  100 -----
** Running simulation at H = 0
** Running simulation at H = 0.1
** Running simulation at H = 0.2
** Running simulation at H = 0.3
** Running simulation at H = 0.4
** Running simulation at H = 0.5
** Running simulation at H = 0.6
** Running simulation at H = 0.7
** Running simulation at H = 0.8
** Running simulation at H = 0.9
** Running simulation at H = 1
** Running simulation at H = 1.1
** Running simulation at H = 1.2
** Running simulation at H = 1.3
** Running simulation at H = 1.4
```

[17]:
```
def m(h_over_J,J,T):
    h = array(h_over_J)*J
    beta = 1.0/T
    return exp(beta*J)*sinh(beta*h)/sqrt(exp(2*beta*J)*sinh(beta*h)**2 +␣
 ↪exp(-2*beta*J))

h_plot = np.linspace(0.0, 1.4, 200)
plot(h_plot, m(h_plot,1.0,1.0),lw=3.0,alpha=1.0,label='Analytic solution')

for size,m_of_h in zip(sizes,ms_of_h):
    plot(array(hs),m_of_h,marker='s',ls='--',label=str(size))

legend()
grid()
xlabel(r'H / J')
ylabel(r'm')
title("T = 1.0");
```


T = 1.0
m
H / J
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Analytic solution
2
5
10
50
100