Consider a lattice of N spins with pairwise interactions described by the Ising model,

$$H = -J \sum_{i,j \text{ N.N.}} s_i \cdot s_j \ , \tag{1}$$

where each spin can take values $s_i = \pm 1$, and where the sum ranges over pairs of spins that are nearest neighbors.

In this exercise, we will calculate the magnetization and energy of a system of spins described by the Ising model, using numerical Monte Carlo integration. Monte Carlo integration is a powerful numerical technique to calculate e.g. quantity averages. It is based on sampling at random and averaging the quantity of interest over all the samples.

We will actually use an "improved" variant of Monte Carlo integration, which is called the Metropolis algorithm. This algorithm biases the samples towards the regions that contribute more to the integration. More background on these algorithms can be found, e.g., on Wikipedia.[1][2] The algorithm, applied to our case, consists in the following steps:

1. Start with a completely random initial state of the system $\{s_i\}$;
2. Randomly pick a spin i, and compute a proposal for a new state $\{s_i'\}$, defined by flipping the spin i on the system $\{s_i\}$;
3. Compute the ratio a of the "weight" of the state $\{s_i'\}$ and the "weight" of the state $\{s_i\}$;
4. Accept the new state $\{s_i'\}$ randomly with probability a, otherwise stay in the current state $\{s_i\}$. If $a > 1$, always accept the new state;
5. Store the current sample;
6. Repeat steps 2.–5.

The *weight* in step 3. is the (ensemble) probability that the system can be found in the given state. In the canonical ensemble, this probability is known: it's simply $Z^{-1} \exp\left[-\beta H\left(\{s_i\}\right)\right]$.

One *sweep* is defined as N *updates*, i.e., repeating the steps 2.–4. N times. During the Metropolis procedure, it might take some time for the system to find the region where the states contribute most to the average ("thermalization"). One should discard the first $n_{\text{thermalize}}$ sweeps, during which the system thermalizes. Then, one should keep one sample per sweep for a additional n number of sweeps. One should play around to get good values for $n_{\text{thermalize}}$ and n.

The output of the algorithm is then simply the average of all the stored samples.

Exercise 1. *Numerical Simulation of the Ising Model.*

[counts as 3 exercises; points are given proportionally to fraction of contents done if partially solved]

(a) For a one-dimensional lattice with periodic boundary conditions, write a program that calculates the average magnetization $\langle |m| \rangle$ and average energy $\langle H \rangle$ of the system. Use the programming language of your choice (we recommend `python` with `numpy`[3], but MATLAB is also good).

[1] `http://en.wikipedia.org/wiki/Monte_Carlo_integration`
[2] `http://en.wikipedia.org/wiki/Metropolis-Hastings_algorithm`
[3] `http://www.numpy.org/`

Display a graph of the magnetization and average energy as a function of inverse temperature β. Do you notice any critical behavior?

Hints. Use the following parameters. number of sites: $L = 20$*, coupling:* $J = 1$*,* $n_{\text{thermalize}} = 1000$*, number of sampling sweeps:* $n = 4000$*. Choose* β *values ranging from* $0.1 \ldots 3.5$*.*

Also, it may be useful to display the energy and magnetization per spin*, in case you wish to compare graphs for different values of* L*.*

It is always important to perform correct error analysis on the Monte Carlo simulation. While an in-depth treatment would be the topic of a computational physics lecture (we specifically refer to the lecture by Prof. M. Troyer), we will do a naive estimate of our errors by using the formula valid in Monte Carlo integration of a quantity X based on n samples X_k:

$$\Delta X \approx \frac{1}{\sqrt{n-1}} \sqrt{\frac{1}{n} \sum X_k^2 - \frac{1}{n} \left[\sum X_k \right]^2} \tag{2}$$

(b) Include naive error bars on your graphs from point (a) using formula (2).

(c) Adapt your program to handle a two-dimensional Ising model, with the spins arranged in a square lattice of $L \times L = N$ sites. Each spin has four nearest neighbors, with periodic boundary conditions.

Display graphs of how the energy and magnetization evolve as you update the system. Estimate how many sweeps the simulation needs to thermalize.

Redraw the graphs again from point (a), with error bars. Observe the critical behavior at

$$T_c = \frac{2J}{1+\sqrt{2}} \approx 2.69 \qquad \text{i.e.} \quad \beta \approx 0.44 \; . \tag{3}$$

Hint. If your code runs too slowly, adapt the length L*, the number of sweeps, and the number of* β *values that you compute for accordingly.*

(d) Run your program for various lattice sizes L, and compare the graphs. What do you observe?

Tips. *Each possible configuration of the system is described by the states of all spins, i.e. by the set of numbers* $\{s_i\}$*. The energy of a given spin configuration is* $E = H(\{s_i\})$*, and its magnetization is* $m = \sum_i s_i$*. Our task is to compute the averages of these quantities for a system at a finite temperature* T *(or equivalently, at finite inverse temperature* $\beta = 1/T$*). The canonical ensemble is relevant here, i.e. the average of a function* f *is computed as*

$$\langle f \rangle = \frac{1}{Z(\beta)} \int_\Gamma f(x)\, \mathrm{e}^{-\beta H(x)}\, \mathrm{d}x = \frac{1}{Z(\beta)} \sum_{\{s_i\}} f(\{s_i\})\, \mathrm{e}^{-\beta H(\{s_i\})} \quad ; \quad Z(\beta) = \sum_{\{s_i\}} \mathrm{e}^{-\beta H(\{s_i\})} \; . \tag{T.1}$$

To calculate this average using the Metropolis algorithm, one implements steps 1-6 given above. Specifically:

1. *Start with a completely random spin configuration. That is, each spin is chosen up or down with probability* 1/2*.*
2. *Choose at random a site* i*, at which we will consider flipping the spin. Let* $\{s_j\}$ *be the spin configuration before flipping and* $\{s'_j\}$ *be the configuration after flipping the* i*th spin, i.e.* $s'_j = s_j$ *for* $j \neq i$ *and* $s'_i = -s_i$*.*

3. *We now need to calculate the probability with which we will accept the new state. This is given by*

$$a = \frac{\mathrm{Prob}\left(\{s_i'\}\right)}{\mathrm{Prob}\left(\{s_i\}\right)} = \frac{Z^{-1}\,\mathrm{e}^{-\beta H(\{s_i'\})}}{Z^{-1}\,\mathrm{e}^{-\beta H(\{s_i\})}} = \mathrm{e}^{-\beta\,\Delta E} \quad ; \quad \Delta E = H\left(\{s_i'\}\right) - H\left(\{s_i\}\right) \ .$$

   Note that in the calculation of ΔE, you should avoid recalculating from scratch the energy of both configurations; it suffices to think and figure out the change in energy caused by flipping the spin i.

4. *At this point, we need to accept the new state with probability a. This can be done by drawing a random number r uniformly between zero and one, and accepting the new state only if $r \leqslant a$. Remember that if $a > 1$, we always accept the new state. To* accept *the new state means to change the current state $\{s_i\}$ to the new state $\{s_i'\}$. If the new state was not accepted, we stay in the current state.*

5. *Instead of storing each update as a sample, we will run a full sweep between each samples. (This produces better results.) That is, we run N times the steps 2.–4. before storing the current value of interest as a sample.*

6. *We repeat n times the steps 2.–5. above (which correspond to a sweep and storing a sample), in order to collect n samples. Note that, we should first run $n_{\mathrm{thermalize}}$ sweeps without storing any samples, to let the system thermalize. Equivalently, we should run $n_{\mathrm{thermalize}} + n$ times the steps 2.–5. above, and discard the first $n_{\mathrm{thermalize}}$ samples.*

In the end, the Monte Carlo Metropolis estimate of the average magnetization is simply

$$m_{M.C.} = \frac{1}{n}\sum_k m_k \ ,$$

where the m_k are the magnetization of each of the samples stored in point 5. The same holds for the average energy. Naive error bars on this quantity are provided by the formula (2)*.*

Note, additionally, that you do not need to recalculate the energy of the full system at each update. To optimize your code, simply work out the energy difference due to flipping one spin, and update an energy variable accordingly. The same holds for the magnetization.

Working with `python` and `numpy`. *Python is a very versatile and powerful language, which can also be used for numerics. To get started with* ***`python`*** *with* ***`numpy`****, we provide a very basic example below of a program that calculates the volume of a half-sphere by very basic Monte Carlo integration (note that this is not the Metropolis algorithm).*

If you're new to this language, or if you're looking for a specific function, you should read through the tutorials and documentations for ***`python`***[5]*,* ***`numpy`***[6] *and* ***`matplotlib`***[7]*. Also always remember that when searching for a function, Google is always your friend!*

[5] `http://docs.python.org/2.7/library/`; good tutorial at `http://docs.python.org/2/tutorial/`
[6] `http://docs.scipy.org/doc/numpy-1.7.0/reference/`
[7] `http://matplotlib.org/users/`; tutorial at `http://matplotlib.org/users/pyplot_tutorial.html`

```
import numpy as np
from numpy.random import random, randint, rand
import matplotlib.pyplot as plt

# Calculate the volume of a half-sphere using Monte Carlo (not Metropolis) integration.
# We will numerically estimate the integral \int dxdy \sqrt(R^2 - (x^2+y^2)) by sampling
# points at random, and noting down the value of the function at those points. Averaging
# all the values gives an estimate to the mean function value, which is directly related
# to the volume by a factor which is the surface of the disc of radius R in the XY plane.

class HalfSphereMC:
    """
    Perform some Monte Carlo Integration
    """
    def __init__(self, radius):
        # store the radius of the sphere
        self.radius = float(radius)
        # start at a random point on the X-Y plane
        self.state = self.sample_random_point()

    def sample_random_point(self):
        s = [2*self.radius*(random()-0.5), 2*self.radius*(random()-0.5)]
        # retry until we are inside the disc
        while (s[0]*s[0] + s[1]*s[1] > self.radius*self.radius):
            s = [2*self.radius*(random()-0.5), 2*self.radius*(random()-0.5)]
        # return this point
        return s

    def value(self):
        """Returns the value of the function at this point"""
        return np.sqrt(self.radius*self.radius - (self.state[0]*self.state[0]
                                                  + self.state[1]*self.state[1]))

    def update(self):
        """
        Perform Monte Carlo update: sample a new point inside the disc. To sample inside
        the disc, we sample a random point and retry as long as we are out of the disc
        """

        # sample a new random point
        newstate = self.sample_random_point()
        # and update our state
        self.state = newstate


# Some parameters
R = 1.0
nsamples = 5000

# Create a HalfSphereMC object
sim = HalfSphereMC(R)

# prepare an empty array of size nsamples
samples = np.empty( [nsamples] )
# store, for illustration purposes, the sample points
samples_xy = np.empty( [nsamples,2] )
```

```
# repeat nsamples times, i=0,1,...,(nsamples-1)
for i in range(nsamples):
    sim.update()
    # store the current sample
    samples[i] = sim.value()
    samples_xy[i] = sim.state

# need to normalize integral by the surface of the integration region
BaseSurf = np.pi*R*R
# average out for the volume
Vol = np.sum(samples) / nsamples  * BaseSurf

print "The volume of the half-sphere of radius R=%f is Vol=%f." %(R, Vol)

# NOTE: No error analysis is provided here, for the purpose of this sample code. But it
# is IMPORTANT in Monte Carlo simulations.

# Show a figure of the sample points

plt.figure()
plt.plot(samples_xy[:,0], samples_xy[:,1], 'b+')
plt.xlabel(r'X')
plt.ylabel(r'Y')
plt.savefig('samples_disc.pdf')

plt.show()
```