# Chapter 2
# Two-dimensional Ising Model - Metropolis-Hastings's algorithm

**Abstract** This chapter deals with the most commonly used Monte Carlo algorithm to calculate Ising model estimations. This is the opportunity to cover the following concepts:

- *Physics*: The Ising model constitutes the archetypal model of classical statistical physics. The phase transition in the two-dimensional Ising model is briefly presented.
- *Algorithm*: Stochastic methods are illustrated. Monte Carlo methods are introduced via the Metropolis-Hastings algorithm.
- *Computer science*: This chapter requires the generation of random numbers. The library `numpy.random` will be presented. An introduction is provided to the use of objects to describe physical systems.

## 2.1 The Ising model

The Ising model (1920) is described by the energy of a configuration of spins on a lattice, described by the Hamiltonian

$$\mathscr{H} = -J \sum_{\langle i,j \rangle} \sigma_i \cdot \sigma_j - \mu \sum_i h_i \sigma_i, \tag{2.1}$$

where $\sigma_i = \pm 1$ is a localized *spin* on the site $i$ of a lattice, $J$ denotes a coupling constant, $\mu$ denotes the magnetic moment and $h_i$ the magnetic field on the site $i$. The bracket notation $\langle i, j \rangle$ indicates that $J$ only couples neighboring spins (with periodic boundary conditions).

This model plays a very important role in classical statistical physics. It allows investigations of various topics, as e.g., magnetism, binary alloys, the liquid-vapor transition, opinion dynamics, etc.

We will consider the case of ferromagnetism $J > 0$. In this case, aligned spin configurations are favored, i.e., configurations in which adjacent spins are of the

same sign have higher probability. The second term in the Hamiltonian describes how spins interact with the magnetic field. Spins tend to line up in the direction of the field.

### 2.1.1 Phase transition

A physical system undergoes a phase transition when a spontaneous symmetry breaking occurs in the absence of external field ($h = 0$). It is possible to show that the Ising model does not undergo any phase transition for a dimension $d = 1$, but it does undergo a phase transition at finite temperature for a dimension $d \geq 2$.

A phase transition is characterized by an order parameter. For instance, the order parameter for the Ising model is the magnetization defined as the average value of the spin:

$$\langle m \rangle = \lim_{N\to\infty} \frac{1}{N} \sum_i \langle \sigma_i \rangle = \begin{cases} = 0 & \text{if} \quad T \geq T_c \\ \neq 0 & \text{if} \quad T < T_c \end{cases} \tag{2.2}$$

The critical temperature $T_c$ is defined by the temperature above which the spontaneous magnetization vanishes. At critical temperature, the magnetic susceptibility $\chi$, the specific heat $c_V$ and the correlation length $\xi$ diverge and determine the critical exponents $\alpha$, $\gamma$, $\nu$, $\beta$, $\delta$ and $\eta$ according to the scaling laws:

$$c_V \simeq (T - T_c)^{-\alpha}, \qquad \chi \simeq (T - T_c)^{-\gamma}, \qquad \xi \simeq (T - T_c)^{-\nu}, \tag{2.3}$$

$$\langle m \rangle \simeq (T_c - T)^{\beta}, \qquad \langle m \rangle \simeq h^{1/\delta}, \qquad \langle \sigma(0)\sigma(r) \rangle \simeq r^{-d+2-\eta}. \tag{2.4}$$

Onsager showed that for $d = 2$, the critical exponents are $\alpha = 0, \quad \beta = 1/8, \quad \gamma = 7/4, \quad \eta = 1/4, \quad \nu = 1, \quad \delta = 15$

### 2.1.2 Calculation of physical quantities

We consider a system of $N = L \times L$ sites in a constant external field $h$. All physical properties at the temperature $k_B T = 1/\beta$ can be obtained from the partition function

$$Z = \sum_{\sigma_1,\ldots,\sigma_N} \exp\big(-\beta E(\{\sigma_1, \ldots, \sigma_N\})\big),$$

by means of the following relations:

$$\langle m \rangle = \frac{1}{\beta N} \frac{\partial \ln Z}{\partial h}, \qquad \chi = \frac{\partial \langle m \rangle}{\partial h} = \beta N \Big( \langle m^2 \rangle - \langle m \rangle^2 \Big)$$

$$\langle e \rangle = \frac{-1}{N} \frac{\partial \ln Z}{\partial \beta}, \qquad c_V = \frac{\partial \langle e \rangle}{\partial T} = k_B \beta^2 N \Big( \langle e^2 \rangle - \langle e \rangle^2 \Big)$$

To calculate these physical quantities, it *is sufficient* to calculate a sum of $2^N$ terms! In practice, it is very difficult to do the explicit calculation for $N \geq 16$. This motivates the use of **Monte Carlo methods** for simulations of the Ising model.

Monte Carlo methods refer to a broad class of computational algorithms that rely on repeated random sampling to numerically solve physical and mathematical problems when it is impossible to use other approaches. They are often used to simulate systems with many coupled degrees of freedom (fluids, disordered media, interacting particle systems, kinetic models, etc), phenomena with large uncertainty in inputs (risk evaluation, failure predictions, cost/schedule overruns), numerical evaluation of multidimensional definite integrals with complicated boundary conditions, and to solve any problem having a probabilistic interpretation by building an appropriate Markov chain associated with probability distributions that possibly follow an evolution model.

## 2.2 How can we compute $\pi$ by throwing stones?

In this section, we illustrate the spirit of Monte-Carlo methods by a classical example: the numerical estimation of $\pi$.

A very simple way to calculate $\pi$ by means of a stochastic method relies on the random sampling of the surface of a circle of radius unity contained within a square of side length equal to 2 unit lengths. Stones are thrown randomly within the square, in a uniform way. The ratio of the number of stones found within the circle to the number of stones out of the circle gives in principle an estimation of $\pi$ since the number of stones within each surface is proportional to the corresponding surface. Hence, the stone number ratio is equal to the ratio of the surface of the circle to the surface of the square, i.e., $\pi/4$. Thus, estimating $\pi$ amounts to multiplying by four the surface ratio of the circle to the square, which can be approximated by the fraction of stones within the circle.

Mathematically, the i$^{\text{th}}$ stone throw is described by a random variable $X_i$, equal to 1 if the stone falls within the circle and to 0 if it falls outside the circle. The average, rms value and standard deviation for $X_i$ are calculated as:

$$\langle X_i \rangle = \frac{\pi}{4} \cdot 1 + \frac{4-\pi}{4} \cdot 0 = \frac{\pi}{4}, \qquad \langle X_i^2 \rangle = \frac{\pi}{4}, \qquad \sigma_{X_i} = \frac{1}{4}\sqrt{4\pi - \pi^2},$$

giving the features of the distribution function for $X_i$. The average indicates that the distribution of throws will be centered on $\pi/4$ and the standard deviation indicates how much a throw departs from the average value, on the average. The finite value of the standard deviation $\sigma_{X_i} \sim 0.4$ makes it impossible to obtain an accurate estimation with a single throw. Accuracy improves if many throws are performed and the result is averaged.

In this aim, a new random variable is introduced. The stone is thrown $M$ times and the empirical average $X$ is calculated as:

$$X = \frac{1}{M}\sum_{i=1}^{M} X_i, \qquad \langle X\rangle = \frac{1}{M}\sum_{i=1}^{M}\langle X_i\rangle = \frac{\pi}{4}$$

An estimation of $\pi$ is thus obtained by multiplying by four this new random variable, the empirical average over $M$ throws.

How many throws do we need to obtain a good estimation? To answer this question, we evaluate the error bar on the empirical average:

$$\langle X^2\rangle - \langle X\rangle^2 = \frac{1}{M^2}\sum_{i,j=1}^{M}\langle X_i X_j\rangle - \frac{1}{M^2}\sum_{i,j}\langle X_i\rangle\langle X_j\rangle = \frac{1}{M}\sigma_{X_i}^2 \Rightarrow \quad \boxed{\sigma_X = \frac{\sigma_{X_i}}{\sqrt{M}}}$$

The result shows that the squared standard deviation of the average over $M$ throws is equal to the ratio of the squared standard deviation for a single throw to the number of throws $M$. A good estimation is expected if the number of throws is large as the standard deviation of $X$ decreases as $M^{-1/2}$, when the number of stone throws increases. A large number of throws is expected to lead to a small standard deviation, i.e., a good estimation of the average with a small error bar. In practise, several thousands of throws are necessary to reach a reasonably good estimation of $\pi$ with two digits accuracy.

The spirit of a Monte-Carlo method is therefore to use random numbers to perform a stochastic integration for a given problem rather than performing an exact integration. Accompanying a stochastic integration with an evaluation of the statistical error is always a valuable information to assess the accuracy of the estimation.

## 2.3 Calculation of integrals with the Monte Carlo method

This strategy can be used to calculate other integrals. For instance, Figure 2.1 represents a function $f(x)$. We wish to calculate the quantity $I = \int f(x)dx$ stochastically.

In this aim, we can generate random numbers $x_i$, say with a uniform distribution function between $-4$ and 4, calculate the value $f(x_i)$ and sum the results. By repeating this sequence and averaging the results, an approximation of the integral $I$ will be obtained. However, this method is not very efficient due to the choice of a uniform distribution function between $-4$ and 4 for sampling the function. The function is often sampled in areas with negligible values of the function, which therefore does not contribute much to the result.

In order to be efficient, we wish to sample the $x$-axis in a way that is compatible with the function, i.e., we wish to use a better distribution of random draws so as to have more draws where the function is large and less where it is negligible. The use a probability law that favors $x$ where $f(x)$ is large will achieve this goal and is called *Importance sampling*. It is possible to show that the standard deviation of the final result will diminish if advantage is taken of importance sampling.

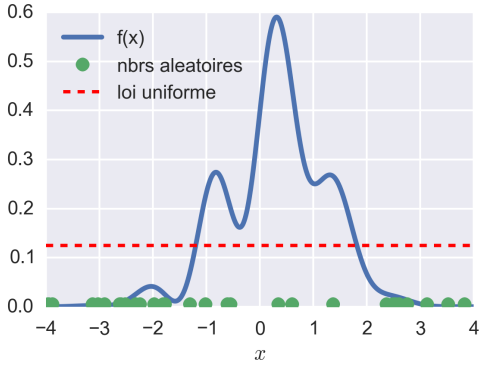


**Fig. 2.1** Stochastic evaluation of the function $f(x)$.

### 2.3.1 *Importance sampling*

To calculate the integral $I$ stochastically, we are looking for a distribution $\rho(x)$ to sample the integral:

$$I = \int f(x)dx = \int \frac{f(x)}{\rho(x)}\rho(x)dx \simeq \frac{1}{M}\sum_{i=1}^{M}\frac{f(X_i)}{\rho(X_i)} \tag{2.5}$$

The integral is rewritten by using the ratio $g(x) = f(x)/\rho(x)$ so as to obtain an integral of the function $g(x)$ over the distribution of points $\rho(x)$. In the language of statistics, it means that the random variables $X_i$ are distributed following the density distribution $\rho(x)$ and for each variable, the function $g(X_i) = f(X_i)/\rho(X_i)$ are calculated. The question is which law to use for $\rho(x)$?

To minimize the standard deviation in the empirical estimation of the integral, the best is to find a law that follows the behavior of $f(x)$. Here we could take a normal distribution for $\rho(x)$ such as that plotted in dashed line on Fig. 2.2

Ideally we would like to be able to generate distributed random numbers according to a law $\rho(x) \simeq f(x)$. In this case, the draw distribution is $f(x)$ and Eq. (2.5) shows that the empirical evaluation of $I$ amounts to adding one $(f(X_i)/f(X_i))$ for each draw that falls where $f(X_i)$ is non negligible. This is the basis of the **Metropolis-Hastings** algorithm.

### 2.3.2 *Markov chain and stationary distribution*

Our objective is to generate random numbers distributed according to an arbitrary density law $\rho(x)$. How can this goal be achieved?

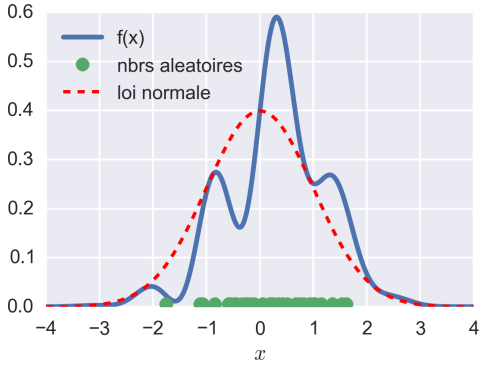


**Fig. 2.2** Stochastic evaluation of the integral $I = \int f(x)dx$ using importance sampling.

For a uniform law, we can figure out that this task is simple. The numpy library in python contains random number generators that effectively provides random numbers with a uniform distribution over a given range. Mathematical techniques exist to obtain a normal distribution from a sequence of uniformly distributed random numbers, hence the case of a normal distribution is simple as well. In contrast, for an entirely general law $f(x)$, this task is not easy.

Metropolis and Hastings have resolved this problem and found how to generate random numbers distributed according to a predetermined law. The objective is to create a Markov chain

$$x_1 \to x_2 \to x_3 \to \dots$$

that will be distributed according to a certain law $f(x)$. In a Markov chain, a number is generated only from the present number and not the previously generated numbers. This is called the Markov property: there is no memory effect; the probability of moving to the next state depends only on the present state and not on the previous states. The value of $x_3$ will depend only on the value of $x_2$ and not on the value of $x_1$.

To define a Markov chain, we need to define the probability $W_{xy}$ of moving from $x$ to $y$. The Markov chain will be constructed recursively by using this probability of moving from a value to the next. From this information, it is possible to calculate the probability at step $n+1$ to obtain the value $x$, i.e., the probability distribution at step $n+1$. There are two ways to obtain $x$, either from a value $y$ different from $x$ at the previous step and a transition from $y$ to $x$, or from the value $x$ at the previous step and an absence of transition to a different value. Hence for the first way, $P_{n+1}(x)$ is equal to the probability $P_n(y)$ to start with a value $y$ different from $x$ multiplied by the probability $W_{yx}$ to move from $y$ to $x$. To the latter result must be added the probability to preserve the value $x$ which is equal to the probability $P_n(x)$ to have the value $x$ at step $n$ multiplied by the probability $(1 - \sum_{y \neq x} W_{xy})$ to not transit to a different value:

$$P_{n+1}(x) = \sum_{y \neq x} W_{yx} P_n(y) + \left(1 - \sum_{y \neq x} W_{xy}\right) P_n(x) \tag{2.6}$$

This defines the probability distribution $P_{n+1}(x)$ of the state $x$ at step $n+1$.

If this operation is repeated many times, we can expect that the Markov chain will eventually converge to a stationary distribution. The probability distribution will therefore no longer change. At the next step, the probability distribution of the generated numbers will be the same as that at the previous step. If this distribution becomes stationary, $P_{n+1} = P_n = P$, hence Eq. (2.6) becomes

$$P(x) = \sum_{y \neq x} \left(W_{yx} P(y) - W_{xy} P(x)\right) + P(x),$$

which leads to the condition (2.7) over the probability distribution called the *balance* equation

$$\boxed{\sum_y W_{yx} P(y) = \sum_y W_{xy} P(x)}. \tag{2.7}$$

It is not our purpose to establish mathematically the conditions to be satisfied for the convergence of the probability distribution to a stationary distribution. In fact, this convergence is not true for any Markov chain or for any transition probability $W_{xy}$. Certain conditions must be satisfied. Among these, ergodicity must be checked, i.e., any value $y$ must be reachable from any value $x$ in a finite number of transitions. Mathematically, it is possible to show that this distribution becomes stationary for an ergodic system satisfying a set of additional conditions. *a few more details?*

### *2.3.3 The Metropolis-Hastings algorithm(1953)*

We have seen that we must chose a transition probability $W_{xy}$ satisfying the balance equation (2.7) if we wish to obtain the probability distribution $P$. If we find a transition probability satisfying the balance, we will construct in principle a distribution that converges toward a stationary distribution. How to find such a transition probability?

We will in fact require a stronger condition called the *detailed balance*: Rather than satisfying the balance, we will require that a term by term equality be satisfied in Eq. (2.7). If Eq. (2.7) is true term by term, the sum is certainly true. It is therefore sufficient to find a transition probability $W_{xy}$ which satisfies the *detailed balance*

$$W_{yx}\,\rho(y) = W_{xy}\,\rho(x)$$

This condition is stronger than necessary but if it is satisfied, the balance will be satisfied as well and we will converge toward a probability density $\rho$.

The idea of Metropolis-Hastings's algorithm is to generate the Markov chain in two stages:

1. Propose a change $x \to y$ with transition probability $T_{xy}$ that can be freely chosen. This change is proposed but will not be systematically accepted.
2. Accept this change with probability $A_{xy}$. From time to time, the change will be accepted and then, we move to state $y$. Otherwise, the change is refused and we stay with $x$ at the next step of Markov's chain.

The transition probability $T_{xy}$ can be chosen (almost) arbitrarily; *almost* refers to the condition that the choice of $T_{xy}$ must allow the chain to be ergodic (and other required conditions to be satisfied, if any) to reach a stationary distribution.

As a last task, we need to find the acceptance probability $A_{xy}$ in order to satisfy the detailed balance for $W_{xy} = T_{xy}A_{xy}$.

**Metropolis-Hastings** show that the acceptance probability can be chosen as

$$W_{xy} = T_{xy}A_{xy}, \qquad A_{xy} = \min\left(1, \frac{\rho(y)T_{yx}}{\rho(x)T_{xy}}\right), \tag{2.8}$$

Equation (2.8) fully describes if the proposed change is accepted or not: the probability to transit from $x$ to $y$ is equal to the probability that this move be proposed multiplied by the probability $A_{xy}$ that this change be accepted. It is possible to check a posteriori that the detailed balance is satisfied by introducing (2.8) into it. The ratio $\rho(y)T_{yx}/\rho(x)T_{xy}$ will indeed appear on the right hand side of the detailed balance, and its inverse on the left hand side. On the right hand side, this ratio will be larger than 1 (if not, its inverse will be larger than 1 and the same reasoning can be performed with the left hand side), thus we will obtain $T_{xy}\rho(x)$. On the left hand side, the inverse ratio will be smaller than 1, thus the minimum is equal to this ratio and the quantity $\rho(y)$ cancels out. The transition probability $T_{yx}$ also cancels out and we are left with $T_{xy}\rho(x)$ leading to trivial identity.

To summarize, Metropolis and Hastings strategy to distribute a variable according to a certain law consists in a two stage approach: a free proposal for a change in the variable followed by an acceptance with probability defined by (2.8), which involves the law that we wish to generate.

## 2.4 Illustration of Metropolis-Hastings's algorithm

Metropolis-Hasting's algorithm is very simple to code, as will be illustrated by an example, for instance for the function displayed in Fig. 2.3. Let us try to distribute numbers with $\rho(x) = f(x)$. The first choice to make is how to transit from a variable $x$ to a variable $y$?

We choose a very simple law: A value $x$ can undergo a transition either to $x+\delta$ or to $x-\delta$, each with a probability $1/2$. The transition probability is thus always zero, $T_{xy} = 0$, except for the transition from $x$ to $x \pm \delta$: $T_{x,x\pm\delta} = 1/2$.

Clearly $T_{xy} = T_{yx}$, hence the transition probabilities cancel out in the fraction defining $A_{xy}$ (see Eq. (2.8)), hence the probability to accept the proposed change is the minimum between 1 and the ratio $f(y)/f(x)$:

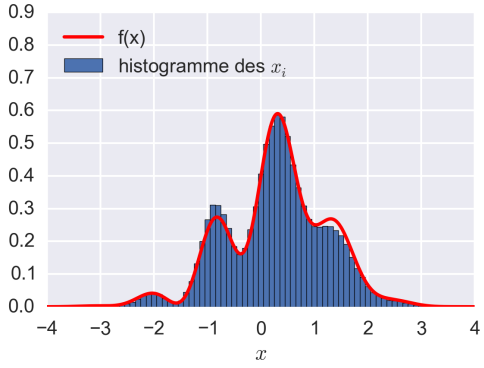


**Fig. 2.3** Illustration of Metropolis-Hastings's algorithm. The function $f(x)$ is integrated stochastically.

$$A_{xy} = \min\left(1, \frac{f(y)}{f(x)}\right). \tag{2.9}$$

Metropolis-Hastings's algorithm can be implemented in the following short code.

```
1  x, M, delta = 0.0, 100000, 0.1
2  h = np.zeros(M)
3
4  for i in range(M):
5      # Propose new x
6      new_x = x + delta*rnd.choice([-1,1])
7      A = f(new_x) / f(x)
8      # Accept proposal with prob A
9      if (rnd.random() < A): x = new_x
10     h[i] = x
11
12 plt.hist(h, np.arange(-4.05,4.05,0.1),
13          normed=True)
```

Metropolis algorithm

The first line initializes a set of variables. We start with $x = 0$. We will perform $M = 10^5$ steps and the spacing between $x$-values is set to $\delta = 0.1$.

In the loop starting at line 4, the Markov chain is generated with $M$ values of $x$. The new value for $x$ is equal to the old one plus $\delta$ multiplied by plus or minus one, with probability $1/2$. This is achieved via the numpy library and the function rnd.choice(). In this way, a new $x$ is chosen and its value will be $x+\delta$ or $x-\delta$.

In line 7, the probability to accept this change is calculated as $f(y)/f(x)$ where $y$ denotes the new $x$. In principle, Eq. (2.9) requires to take the minimum between 1 and $f(y)/f(x)$ but this is not necessary here.

Line 9 corresponds to the conditional acceptance of the new value of $x$ with probability $A$. This is simply implemented by generating a random number between 0 and 1. If this random number is smaller than $A$, the change is accepted, otherwise, it is rejected. Clearly, if $0 \leq A \leq 1$, this procedure amounts to the acceptance of the new $x$ with probability $A$. Now, it is visible why taking the minimum as in Eq. (2.9) was not necessary in line 7: if $A$ were greater than 1, the minimum between 1 and $A$ would be 1 and the randomly generated number that is smaller than 1 would be smaller than $A$ in any case.

In this way, a Markov chain is generated and values for $x$ are obtained. From time to time, the value of $x$ will be the same as the previous value and now and again it will change. To see how these values are distributed, an histogram is calculated from the table $\mathrm{h[i]}$ storing the long Markov chain of generated numbers. As can be observed on Fig. 2.3, the distribution of generated numbers matches pretty well the function $f(x)$.

Is this algorithm ergodic? The answer is no. In fact, the only values of $x$ that are generated by this algorithm are located on a grid with fixed steps $\delta$, so the algorithm is not ergodic. It does not allow us to generate all possible values for $x$. This could be cured: to generate all possible values for $x$, the algorithm should be modified so as to allow the step size itself to follow a certain distribution. However, this task is left to the reader since this simple code already serves well our purpose of illustrating Metropolis-Hastings's algorithm, will the only difference that generated numbers fall on the points of our initially defined grid.

In summary, the Metropolis-Hastings algorithm is rather easy to implement and rather powerful since it allows us to distribute numbers following any distribution function.

## 2.5 Application of Metropolis-Hastings's algorithm to the Ising model

A way to calculate physical quantities for the Ising model is to calculate stochastically the partition function

$$Z = \sum_{\sigma_1,\ldots,\sigma_N} \exp\big(-\beta E(\{\sigma_1,\ldots,\sigma_N\})\big),$$

where $\{\sigma_1,\ldots,\sigma_N\}$ denotes a spin configuration. In other words, the stochastic calculation of the partition function amounts to generating spin configurations that follow a distribution function given by $Z$. Note that the previous section dealt about the generation of states consisting of a single number following a certain distribution. A generalization is thus necessary to extend the concept to the generation of states with $N$ dimensions.

To this aim, we will generate spin configurations, i.e., sets of $N$ values $\{\sigma_1,\sigma_2,\ldots,\sigma_N\}$ following a probability law given by the Maxwell-Boltzmann dis-

tribution $\rho(x) = \exp\big(-\beta E(x)\big)$. Maxwell-Boltzmann weights can serve to build a suitable distribution because it is positive, it can be integrated (total probability equal to 1), hence we will chose to distribute the configurations according to the law $\exp\big(-\beta E(x)\big)$, where $E(x)$ denotes the energy of the configuration $x$.

Then we build a Markov chain $x_1 \to x_2 \to \dots$ starting from a given configuration, that undergoes changes, etc, using the Monte-Carlo scheme. Each of the configurations $x_i$ is a set of spin values $x = \{\sigma_1, \sigma_2, \dots, \sigma_N\}$. The probability law is the Maxwell distribution $\rho(x) = \exp\big(-\beta E(x)\big)$.

Following Metropolis-Hastings's algorithm, we propose a change of configuration at each step. There are numerous possibilities. A very simple way is typically to reverse a spin on a randomly chosen site. This gives the proposed new configuration, and thus, it defines the transition probability $T_{xy}$. The proposed changes are accepted with the Metropolis-Hastings acceptance rate. From time to time, the spin flip will be accepted; now and again, it will be rejected. Eventually a Markov chain is built that samples spin configurations according to the law given by the partition function.

Once the Markov chain is obtained, physical quantities are easily calculated. For example the average magnetization for site 1 reads as

$$\boxed{\langle m_1 \rangle = \frac{1}{Z} \sum_x \sigma_1 \exp\big(-\beta E(x)\big) = \frac{1}{M} \sum_x^{\mathrm{MC}} \sigma_1(x)}$$

where $\sigma_1$ denotes the spin at site 1, the sum is taken over all configurations and the Boltzmann weight is recognized for configuration $x$ of energy $E(x)$. In the form of a Monte-Carlo sum, the average magnetization at site 1 reads as a sum over the generated Monte-Carlo configurations of the spins at site 1, $\sigma_1(x)$. Since $Z$ serves as the distribution function for the Monte-Carlo process, it is evaluated as $M$ times 1 and is thus simply equal to $M$. However, the Monte-Carlo sum in the numerator denotes the sum of $\sigma_1(x)$ over Monte-Carlo configurations and will in general differ from $M$, leading to an evaluation of the averaged magnetization at site 1.

## 2.6 Comments

- **Choice of the transition probability** $T_{xy}$: There is a relative freedom on the choice of $T_{xy}$. However, a few points are worth highlighting:
  - $T_{xy}$ must be ergodic. This means that any configurationy must be accessible from any configuration $x$. Clearly with the choice made above for the Ising model, this will be the case since it is always possible to select the spins to be flipped in order to achieve the transition between two given configurations.
  - Local changes: With Monte-Carlo methods, there is always a choice between local or extensive (global) changes in the configuration. Both have

advantages and drawbacks. For a large system of spins, if a single spin is flipped in the new configuration, one of the advantages is the easy calculation of the energy of the new configuration since only the interaction energy around the flipped site is modified; the new energy can be calculated by a sum of the only four terms that were modified (interaction with the nearest neighbors). The calculation of the energy will be extremely *fast*. We will be able to very rapidly decide whether this new configuration is acceptable or not, leading to a *high* Metropolis acceptance rate. The disadvantage is that the configuration space is sampled very *slowly* since configurations do not change much with time. A problem appears with the *autocorrelation time*, i.e., many steps are required in order to completely decorrelate two configurations.

  – Global Changes: In contrast, we can chose to propose global changes, i.e., to flip many randomly chosen spins at the same time so as to generate a configuration as different as possible from the previous configuration. This approach samples the configuration space *well*, which is an advantage. The problem of this approach is that by the random selection of the spin flips, the energy of the newly generated configuration is very different from the energy of the previous configuration. It is likely that the proposed change be rejected by Metropolis's algorithm. Eventually, the system will be copied many times and it can take ages before a new configuration is accepted. The disadvantage of the global changes is thus a *slower* calculation of the energy of the new configuration, associated with a usually *lower* Metropolis acceptance rate. A trade-off must be found between these two options.

- **Thermalization time**: We assumed that the distribution generated by the Markov chain will evolve toward a stationary distribution. This is true, however, this will take time. A certain number of steps is necessary before approaching the stationary distribution. This time is called the *thermalization time*. It is rather difficult to evaluate a priori the number of steps required to reach the desired stationary distribution. Nevertheless, before starting the calculation of a mean value, a certain number of Monte Carlo steps must be performed to avoid bias from the choice of an initial configuration, i.e., for the distribution to become stationary. This is called the *warmup procedure*.
- **Decorrelation time**: if we make local modifications, two configurations which follow each other are *correlated*. Hence, it is useless to *measure* (i.e. calculate average values) at every step. In general, for the Ising model, we wait for having done a certain number of steps, a certain number of proposals for a configuration change. For instance, the average magnetization will be measured once per *cycle* of $N$ steps, where $N$ is the number of sites on the network i.e $N = L \times L$, so as to have flipped roughly half of the spins.

## 2.7 Summary: the Monte Carlo loop

In practise, a Monte-Carlo simulation obey the scheme displayed in Fig. 2.4. First

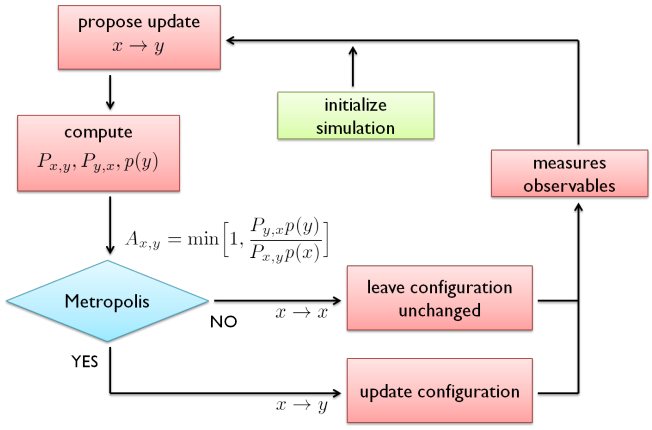


**Fig. 2.4** Monte-Carlo algorithm

the simulation is initialized with an input configuration. Typically, this is a randomly chosen configuration. Then, an update $x \to y$ is proposed. For instance, the flip of a spin somewhere on the network. Next, the probabilities $T_{xy}$, $T_{yx}$ and $\rho(y)$ are calculated. These quantities appear in the formula for the acceptance rate $A_{x,y}$. From the acceptance rate, the proposed update is accepted or rejected. If rejected, the configuration remains unchanged. If accepted, the configuration is modified. In any case, a measurement is performed at this stage, whether the update was accepted or rejected. This is an important point: the correct distribution cannot be obtained if measurements are performed only after an update has been accepted.

**Note**: $P_{x,y} \to T_{xy}$ and $p(x) \to \rho(x)$ (i.e. Fig. 3.4 to be updated)

## 2.8 Generation of random numbers

How do we generate random numbers? This problem is at the heart of a stochastic approach. The fact this task is challenging is illustrated by quoting von Neumann:

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin – John von Neumann

There is indeed no mathematical tool to generate a *true* sequence of random numbers since a formula would be needed – we call it the random number generator – to define what is the next random number in the sequence.

There are plenty of ways to define a *pseudo* random number generator. One of them is the *linear congruential generator* presented below both to illustrate the concept and for the reason that it has a historical importance. However, the reader is warned that it is a very bad random number generator that is not used in practise.

The *linear congruential generator* reads as:

$$X_{n+1} = (aX_n + c) \mod m$$

with e.g. $m = 2^{32}$, $a = 1664525$, $c = 1013904223$.

The idea of the *linear congruential generator* is that we start with a certain integer number $X_n$. It is multiplied by a large number $a$. Another large number is added and the mod with a very large number is finally taken to get the next random number $X_{n+1}$.

These are not true random numbers since at each step, $X_{n+1}$ is known from the previous value $X_n$ and the generator. However, the sequence of pseudo-random numbers gives the impression of a random number sequence for an observer who does not know the algorithm and only sees the sequence.

$X_0$ is the root (or *seed*) of the pseudo-random sequence. If $X_0$ is given, the sequence is entirely determined. This has advantages and drawbacks. On the side of drawbacks, it means that a given seed will lead to the systematic generation of the same sequence of pseudo-random numbers. On the side of advantages, it facilitates bug tracking and correction in a program by reproducing exactly the same sequence of random numbers, and thus the same bug at the same instant (number of steps). In practise, it is healthy to change the seed from time to time in order not to systematically generate the same data sequence.

It is possible to show that this generator gives uniformly distributed numbers. We can bring back the numbers in the range $[0,1]$.

This approach is very simple. Much more sophisticated random number generators exist. A set of well known tests allow scientists to know whether a random number generator is good or not. The linear congruential generator does not pass much of these tests. There are much better generators: in particular, the `numpy` library uses the *Mersenne Twister*, one of the most used random number generator to date. The Mersenne Twister passes sophisticated tests ensuring the absence of correlations between the generated random numbers.

Since the Mersenne Twister is not infallible either, it is a good practise to verify that results of a numerical simulation using sequences of pseudo random numbers remain the same, as far as the statistical observables are concerned, if the random number generator is changed.

In `numpy`, the generation of random numbers is performed with the sublibrary `numpy.random`, loaded by the command on line 1 below. Then, numerous functions can be accessed. A few examples are given on lines 2-5.

```
import numpy.random as rnd
rnd.seed(x)      # set the seed
rnd.random()     # a float in [0,1[
```

```
rnd.choice(seq)  # an element of seq
rnd.randint(a,b) # an integer in [a,b[
rnd.normal()     # a gaussian distr.
```

## 2.9 Animations in python

Data visualization is most often necessary in physics. It may be useful to study data with animations in addition to graphs. For the Ising model, it will be nice to see the configuration changes in the Monte-Carlo chain. The library matplotlib will allow us to animate the results. Here is an example.

```
1  %matplotlib tk
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import matplotlib.animation as animation
5
6  rg = np.arange(-5, 5, 0.1)
7  x, y = np.meshgrid(rg, rg)
8  r2 = x**2 + y**2
9  z = np.sin(r2)/r2
10
11 fig = plt.figure()
12 im = plt.imshow(z)
13
14 def make_frame(t):
15     z = np.sin(r2 + 0.5*t) / r2
16     im.set_array(z)
17     return im
18
19 animation.FuncAnimation(fig, make_frame, blit=False)
```

A python animation

The pertinent library to be imported for animations is found on line 4. Then the animation is called in the following way:

Lines 11-12 build a figure (in this case, an image corresponding to a two-dimensional colored projection of a surface corresponding to the $sinc(r^2)$ function). This figure is the first image of the animation.

Lines 14-17 define a make_frame function that is going to be called successively during the animation and modifies the image at step (time) $t$. Its argument, t, is an integer that starts at 0 and increases as the animation progresses. For instance here,

it redefines the value $z$ of the function at time $t$. On line 16, im.set array updates the values of $z$ in the image. On line 17, make_frame returns an image, im, that will be modified during the animation

To animate, FuncAnimation is called (line 19) with arguments corresponding to the name of the figure fig, the animation function make_frame. To avoid specific problems, it is suggested to add the argument blit=False.

A figure here (or several figs) to show the example.

## 2.10 Using classes

In physics, it is very natural to use classes to describe certain objects. For example, the configuration of spins on the network lends itself well to an implementation in a class.

In Python, a class is defined in the following way. The name of the class is defined by class ClassName (line 1).

```
1  class Configuration:
2      """A configuration of Ising spins"""
3
4      def __init__(self, J, beta, L):
5          """Initialize a rand configuration"""
6          self.size = L
7          ...
8
9      def get_magnetization(self):
10         """Compute the total magnetization"""
11         ...
12         return m
13
14     def get_energy(self):
15         """Compute the total energy"""
16         ...
17         return e
18
19 # my_config is a class instance
20 my_config = Configuration(1., 2.1, 10)
21 print("E = ", my_config.get_energy())
```

Class of an Ising configuration

The member functions in the class are defined as standard functions: e.g. def get_magnetization on line 9, def get_energy on line 14. The unique difference with respect to standard functions is the first argument, which must always be self if the function belongs to a class.

One of the functions is the *Constructor* __init__. It is the function that will construct the instance of the class. Its first argument is also self and it can have any other arguments.

Class variables are always updated by self.some_variable.

a natural way to code a configuration is to generate an instance of the class as indicated on line 20: my_config is a configuration for $J = 1$, a temperature of 2.1 on a lattice of side length 10. Once my_config is created, it is possible to calculate the energy of this configuration by calling the function get_energy using the syntax my_config.get_energy, which returns the energy. my_config.get_magnetization would return the magnetization.

In summary, classes are features by the following four items:

- **Constructor:** this is the method __init__ of the class
- **Methods**: Their first argument is always self
- **Members**: They are accessed with self.xxx where xxx is the variable name
- **Style**: We often use the CamelCase for class names

## 2.11 References


- *Statistical Mechanics*, R.K. Pathria et P.D. Beale, Butterworth-Heinemann.
- *Phase Transitions and Critical Phenomena*, H. Nishimori, Oxford University Press.
- *Introduction to Phase Transitions and Critical Phenomena*, H.E. Stanley, Clarendon Press Oxford.
- *Exactly Solved Models in Statistical Mechanics*, R.J. Baxter, Academic New York.
- *Statistical Mechanics of Phase Transitions*, J.M. Yeomans, Clarendon Press Oxford.

# Tutorial 2
# The two-dimensional Ising model

## Tutorial 2: The two-dimensional Ising model

The Ising model is described by the classical Hamiltonian

$$\mathscr{H} = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j \tag{2.1}$$

where $\sigma_i = \pm 1$ is an Ising spin on the site $i$ of a square lattice and $J > 0$ is a ferromagnetic coupling between nearest-neighbor spins. The square lattice has $N = L \times L$ sites.
Onsager solved this problem analytically and showed that on the infinite lattice ($N \to \infty$) there is a phase transition at a temperature $T_c$ from a disordered state at high temperatures to a magnetic state with finite magnetization $m \neq 0$ where

$$\langle m \rangle = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} \langle \sigma_i \rangle \tag{2.2}$$

The value of the critical temperature (we set $k_B = 1$) is

$$\frac{T_c}{J} = \frac{2}{\log(1+\sqrt{2})} \simeq 2.27 \tag{2.3}$$

The goal of this project is to study the two-dimensional Ising model using a Monte Carlo algorithm and see whether we can find evidences for the phase transition even though we will be dealing with finite systems.

### Warmup: Compute $\pi$ by MonteCarlo

In order to get familiar with the generation of random numbers, you can try to write a Monte Carlo algorithm to compute $\pi$. The idea, as we saw in the lecture, is to start from a circle of radius 1, inside a square box of length 2. You then throw randomly (uniformly) stones in the square box. After having thrown many stones you can compute the ratio of those inside to those outside the circle. This will give you an estimate for $\pi$.
*Challenge*: Show that the error bar on the value of $\pi$ goes like $1/\sqrt{M}$, where $M$ is the number of Monte Carlo steps. To do this, you will need to make several runs for the same fixed $M$ and compute the statistical error bar.

[In]:

## 2.1 Part 1: Set up a class for the Ising model

Let's first start and write a class that will describe the Ising model. The class should have a constructor taking $T$, $J$ and $L$ as parameters and generate a random initial state. The state can be encoded in an $L \times L$ numpy.array. The class could have a structure like

```
class Configuration(object):
    def __init__(self, L, ...):
        self.size = L
        self.spins = ...
        ...
    def get_energy(self):
        ...
        return e
    def get_magnetization(self):
        ...
        return m
```

In order to test your class, you can instantiate it and plot the spin configuration after the initialization. You can use the function below to convert your spin configuration (with +1 and -1) into an image array:

```
def config_to_image(config):
    L = config.size
    im = np.zeros([L,L,3])
    for i,j in itertools.product(range(L), repeat=2):
        im[i,j,:] = (1.,0,0) if config.spins[i,j]==1 else
    (0.,0,0)
```

```
    return im

import matplotlib.pyplot as plt
plt.imshow(config_to_image(config))
```

You can also check your class methods by generating a configuration with all spins up or down and see whether the energy and magnetization are those you expect.

[ ]:

## 2.2 Part 2: The Metropolis move

Here you will write a function that performs a Metropolis Monte Carlo move on the configuration. The function takes an Ising configuration (an instance of the Configuration class) as an argument and has the following signature:

```
def metropolis_move(config):
    ...
```

The function should chose a random site, compute the energy difference between the old configuration and the new configuration where the spin has been flipped and decide whether the move should be accepted or not (using the Metropolis algorithm). Warning: in order to compute the energy difference, only consider the energy change from the links involving the flipped spin. Indeed, all the other links are unchanged and it would be a waste of time to recompute the full energy every time.
How to check? Start from a small lattice, say $4 \times 4$, and call metropolis_move many times at a rather high temperature. This way, you should sample many different configuration. If you energy calculation is right, the energy per spin should always be between -2 and 2.

[ ]:

## 2.3 Part 3: The Monte Carlo simulation live!

Write an animation that shows the evolution of the spins. In order for the animation not to be too slow, the function that is repeatedly called by $\mathrm{FuncAnimation}$ should make many spin flip trials (typically $L \times L$). Otherwise the animation will be really slow. You might want to get more details about the function $\mathrm{matplotlib.animation.FuncAnimation}$ on the matplotlib website.
See how the system behaves for different:

- Temperatures
- System sizes

[ ]:

## 2.4 Part 4: Compute physical averages

The animation above allows to get some insight into the physics of the model. But it is necessary to compute some physical averages to understand more. Because two configurations that only differ by a spin flip are very correlated, one usually waits a certain number of steps before considering a new configuration in the computation of an average. In practice, the Monte Carlo simulation can be cut into cycles: a cycle is a certain number of steps needed to decorrelate the configurations. Also, in the beginning of the simulation, one needs to wait for the Markov chain to reach its stationary distribution. These variables can be used:

- $\mathrm{length\_cycle}$: the number of steps between two *measurements*, i.e. between two configurations that are used to compute physical averages.
- $\mathrm{n\_cyles}$: the number of cycles used to compute the averages. This number corresponds to the number of measurements
- $\mathrm{n\_warmup}$: the number of cycles that are performed in the beginning of the simulation without any measurement. They are done so that the Markov chain reaches a stationary distribution.

You can now write a Monte Carlo simulation:

1. Compute the average magnetization $m$ as a function of the number of Monte Carlo steps on a $20 \times 20$ lattice. This will allow you to judge how many steps are necessary to reach equilibrium. How does this change with temperature and system size?

2. You can then more systematically compute the magnetization, the energy, the magnetic susceptibility and specific heat as a function of the temperature. Start with a rather small lattice $4 \times 4$ and then increase the size. How do these quantities vary?

You will see that these simulations can take some time. Rather than plotting directly after the simulation is done, you might want to save your averages into files and in another cell read the information from these files to do the plots. It is very common to organize a numerical work like this: a production part that generates the data and a postprocessing part to do the plots etc. To save and read a numpy array you can use:

```
np.savetxt("my_file.dat", my_array)
my_array = np.loadtxt("my_file.dat")
```

[ ]:

## 2.5 Part 5: Autocorrelation time

Above we said that configurations that differ only by a spin flip are very correlated and that a certain number of moves are needed to make two configurations independent. As a rule of thumb, we used $L \times L$ updates to disentangle configurations.
However, as we will see here, as one gets closer to the phase transition, it becomes harder and harder to decorrelate two configurations. A way to measure this in the disordered state is to compute the *autocorrelation time* through the quantity

$$\mathscr{C}(t) = \langle m(t)\, m(0) \rangle \tag{2.4}$$

where $m(t)$ is the magnetization of the system at (Monte Carlo) time $t$. If two configurations are completely decorrelated $\mathscr{C}(\sqcup)$ should be zero. Usually $\mathscr{C}(t) \sim \exp(-t/\tau)$ where $\tau$ is the autocorrelation time.
Write a code that computes the autocorrelation time at a given temperature and study it as the temperature is reduced closer to the critical temperature. You can start with an $8 \times 8$ lattice.

[ ]: