LECTURE 10: Monte Carlo Methods II

November 14, 2011

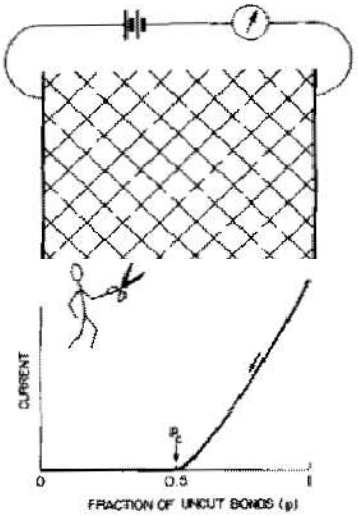
In this chapter, we discuss more advanced Monte Carlo techniques, starting with the topics of percolation and random walks, and then continuing to problems related to statistical physics.

1 Percolation

1.1 Introduction

Percolation is a geometric problem which has played a significant role in statistical physics. In percolation processes, objects are added randomly until an unbroken path spans the entire system (or vice versa, initially unbroken bonds may be cut until connectivity disappears).

As an example, consider a communication network, represented by a very large square-lattice network of interconnections. If the network is attacked by a saboteur who starts to cut the connecting links at random, then what fraction of the links must be cut in order to electrically isolate the two boundary bars? Of course this depends on where you cut the bonds in the net, but if the size of the network goes to infinity, there will be a sharp transition to a disconnected network. The fraction is 0.5 for a 2D square network. If more than half of the links are cut, then the infinite (or very large) network is disconnected.



1.2 Basic percolation models

In the following, we will consider the simplest kind of a percolation problem, namely that of percolation on a square 2D lattice. The lattice is composed of an array of potential occupation sites. Initially the lattice is empty (all sites are unoccupied). Sites are then randomly occupied with probability p.

We now define a rule that nearest neighbour occupied sites are *connected* by a "bond". All mutually connected sites belong to the same *cluster*. The smallest possible cluster is a single site (all neighbouring sites are empty). Percolation occurs when an infinite percolation cluster spans the system; i.e. the cluster extends from one side of the system to the other one. This is called *site percolation.*

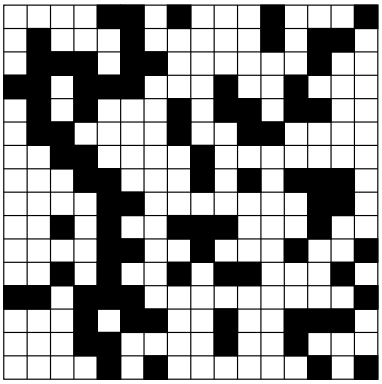


Figure 1: Site percolation. An infinite percolation cluster extends from the top to the bottom of the lattice.

For each value of p, there is a probability P_{span} that a spanning cluster (infinite cluster) will appear. This probability can be determined by generating many realizations of the lattice for each value of p and counting the number of times when a spanning cluster appears. As the lattice size goes to infinity, the probability of having a spanning cluster is zero for $p < p_c$ and unity for $p > p_c$. p_c is called the *critical probability*.

Instead of sites, we can also consider a case where every bond on the lattice is "occupied" (conducting) with probability p. Percolation occurs when "current flows" through the system; i.e. when an infinite percolating cluster appears. This is called *bond percolation*. The critical probability is $p_c \approx 0.59$ for site percolation and $p_c = \frac{1}{2}$ for bond percolation.

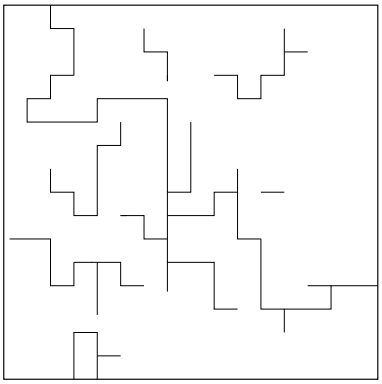


Figure 2: Bond percolation. An infinite percolation cluster extends from the left to the right of the lattice.

1.3 Percolation transition

At p_c the system undergoes a *percolation transition* as a function of p. This is a continuous (geometric) phase transition. Define $n_s(p)$ as the number of clusters per lattice site of size s. Then the probability that **a given** site is occupied and part of a cluster of size s is $sn_s(p)$. The mean cluster size is given by

$$S(p) = \frac{\sum_{s<\infty} s^2 n_s(p)}{\sum_{s<\infty} s n_s(p)}$$

(Because:
The probability that a site belongs to a cluster of size s is $w_s = sn_s / \sum sn_s$ and $S = \sum w_s s$.)

Define $P(p)$ to be the fraction of sites that belong to an infinite percolation cluster. For sparsely occupied lattices, P is zero, but as p increases, we reach the threshold value $p = p_c$ for which $P > 0$. When p is increased further, P continues to grow. Near the percolation threshold ($p \to p_c^-$), the quantity P displays a power law behavior described by the following relation

$$P(p) \sim (p - p_c)^{\beta}$$

where β is called the *critical exponent*. The quantity P is called the **order parameter**.

Geometric phase transitions have a close analogy to real phase transitions which occur in various physical systems. This is the reason why these percolation models have received so much interest in the field of statistical physics.

1.4 Implementation

The implementation of the Monte Carlo method to this problem is quite straightforward. In the case of site percolation, we randomly choose a site on the lattice by choosing two random integers between 0 and $L-1$ (assuming that the sites on the lattice are indexed this way). The selected site is then marked occupied. This process is continued until the desired occupation p is reached. Clusters can then be found by searching for connected pairs of nearest neighbor occupied sites. Finally, we test for percolation by checking if any of the clusters extend from one side of the lattice to the opposite side. The process is then repeated many times in order to obtain an estimate of the probability of encountering an infinite cluster.

An effective way to identify the clusters in a system and to check for the percolating cluster is given by the **Hoshen-Kopelman cluster labeling algorithm**. It is rather easy to identify clusters by going through the lattice row-by-row and labeling each site which is connected to a nearest neighbor by a number. The difficulty arises when we reach a site which connects two previously disconnected segments of a cluster (see Fig. 3). The Hoshen-Kopelman algorithm corrects these mislabelings in a single sweep through the lattice by introducing another set of variables known as the "labels of the labels". In a situation where two segments of a cluster are connected to form one larger cluster, the "label of the label" which is larger is set to the negative of the smaller value (= "true" label of the cluster). In the end, we can identify the two segments as actually being part of the same cluster.

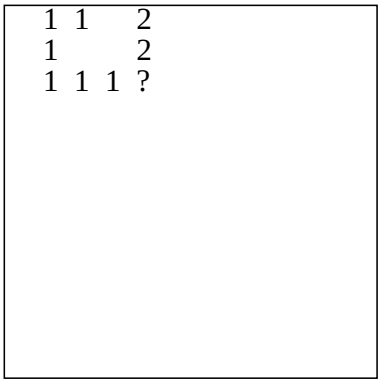


Figure 3: Labeling of the clusters. We sweep through the lattice row-by-row from left to right. Sites with numbers are occupied. The question mark indicates a site which joins previously unattached clusters labeled as 1 and 2.

2 Random walks

A random walk consists of a connected path formed by randomly adding new bonds to the end of the existing walk. The walker can be subject to any restrictions which distinguish one kind of random walk from another. In the following, we will discuss the subject of random walks on a lattice which is a special case of the full class of random walks.

2.1 Simple random walk

Consider a random walker placed on a square lattice at some initial position. At each step, the walker jumps randomly to one of its neighbouring sites. In simple (unrestricted) random walks (RW), the walker may cross the path an infinite number of times with no cost. In a general case of a walk in d dimensions, the end-to-end distance (distance from the initial position to the final position) diverges with the number of steps N according to

$$\langle R^2(N) \rangle \sim N$$

A simulation of the simple random walk can be carried out by picking a starting point and generating a random number to determine the direction of each subsequent step. After each step, the end-to-end distance can be calculated. Errors may be estimated by carrying out a series of independent random walks and performing a statistical analysis of the resulting distribution.

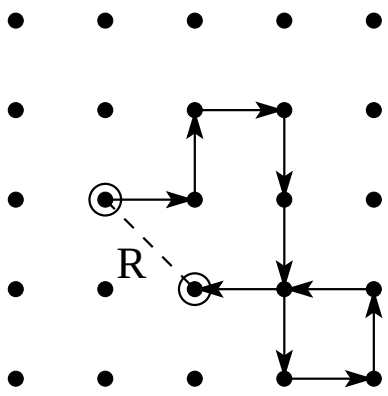


Figure 4: Random walk on a 2D square lattice. R marks the end-to-end distance.

Example: 1D random walk

In a one-dimensional case, the random walker can jump to two opposite directions. If the random walk is *isotropic*, then the walker moves with equal probability either to the left or right. The step length is denoted by l.

Displacement after N steps ($x_0 = 0$)

$$x(N) = \sum_{i=1}^{N} s_i$$

where $s_i = \pm l$.

Displacement squared is given by

$$x^2(N) = \sum_{i=1}^{N} s_i^2$$

This can be easily simulated by generating random numbers r in $[0,1]$ and testing whether $r \leq \frac{1}{2}$ or $r > \frac{1}{2}$.

The average displacement is given by

$$\langle x(N) \rangle = \langle \sum_{i=1}^{N} s_i \rangle = 0$$

The *mean squared displacement* is given by

$$\langle x^2(N) \rangle = \langle \sum_{i=1}^{N} s_i^2 \rangle = \langle \sum_{i=1}^{N} s_i^2 \rangle = l^2 N$$

For the *asymmetric case*, where the probability for a jump to one direction is p and to the other $q = 1 - p$, it is straightforward to show that (start from the binomial distribution)

$$\begin{aligned} \langle x(N) \rangle &= (p-q) l N \\ \langle x^2(N) \rangle &= 4pql^2 N + (p-q)^2 l^2 N^2 \end{aligned}$$

and thus the variance is

$$\langle \Delta x^2(N) \rangle \equiv \langle x^2(N) \rangle - \langle x(N) \rangle^2 = 4pql^2 N$$

If we identify that simulation time as $t = N\tau$, it follows that 1D random walks have

$$\langle \Delta x^2(N) \rangle = 2Dt$$

The quantity D is called the *diffusion coefficient* (of a single random walker). For isotropic 1D walkers

$$D = \frac{l^2}{2} \nu$$

where $\nu = \frac{1}{\tau}$ is the jump rate (frequency).

For anisotropic walkers,

$$D = 2pql^2 \nu$$

2.2 Self-avoiding random walk

An important special type of random walk is the so-called *self-avoiding walk* (SAW). This walk cannot intersect its own path so that it dies when it encounters a configuration where no unvisited neighboring sites are available. At every step, the jump direction is determined at random, but the walker can never go back or cross its own path. Thus *infinite memory* is retained.

For a SAW model, the variance is given by

$$\langle \Delta x^2(N) \rangle \sim N^{2\nu} \qquad (\nu = \frac{3}{4} \text{ for } d = 2)$$

The SAW model has received a lot of interest because this model can be used to model polymers (which are long chainlike molecules that naturally cannot cross themselves). In general, it is difficult or impossible to extract the correct asymptotic behavior for large values of N because it becomes increasingly difficult to generate walks of length N using the Monte Carlo method. This is explained by the fact that the probability that the walk encounters a configuration where it dies increases quickly as N becomes larger.

There are special algorithms which have been designed for the simulation of SAW's. These methods have been applied to solve many interesting research problems, related, for example, to large scale statistical properties of configurations of flexible polymers.

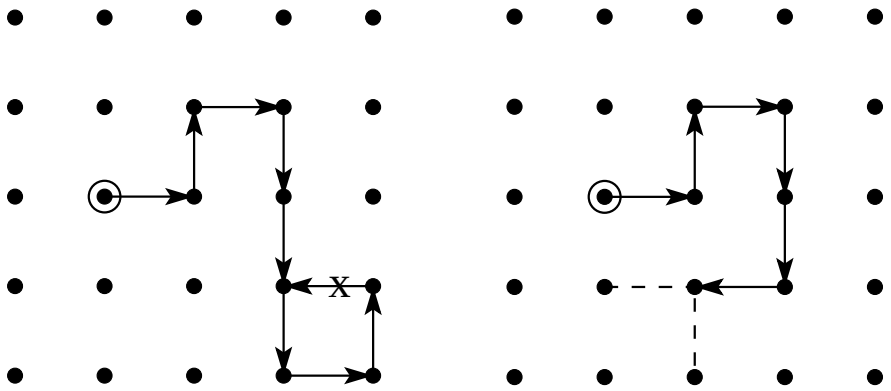


Figure 5: Left: Self-avoiding random walk on a 2D square lattice. ($\times$ marks a forbidden jump.) Right: Growing self-avoiding random walk on a 2D square lattice. (Dashed line marks possible jumps.

2.3 Growing self-avoiding random walk

Because of the problems with generating long SAW walks, alternative strategies have been developed. These 'smart walks' try to find a way to avoid death. An example of such a walk is the growing self-avoiding walk (GSAW). The process proceeds initially as a SAW, but it senses a 'trap' and chooses instead the remaining 'safe' directions so that it can continue to grow. Many similar, still 'smarter' approaches have been developed for simulation of polymeric models.

Random walks and diffusion problems are discussed in more detail in the final chapter of these notes.

3 Monte Carlo simulations in equilibrium

Equilibrium Monte Carlo simulations are based on *generating physical states* of a given system with a desired probability distribution by the use of random numbers.

The MC method is extremely versatile; it applies to e.g.

- classical many-particle systems under equilibrium and non-equilibrium conditions
- quantum and field-theoretic systems
- random systems
- optimization problems

The MC method can be applied to systems with either discrete or continuous degrees of freedom. The basis of the MC simulation method is the theory of **stochastic Markov chains.**

3.1 The Ising model

Before introducing the Metropolis algorithm, we briefly discuss the Ising model which is then used as an example model for the implementation of the Metropolis algorithm.

The Ising model is one of the simplest lattice models, but despite its simplicity this model displays a wide variety of interesting phenomena which have been studied for about three-quarters of a century. We use this model to illustrate how the Monte Carlo method is applied in practice to the study of a statistical physics problem.

We consider the simple Ising model on a square $L \times L$ lattice. The model consists of spins s_i which are confined to the sites of the lattice and may have only the values +1 or -1. These spins interact only with their nearest neighbors on the lattice with interaction constant J. The Hamiltonian of this model is given by

$$\mathcal{H} = -J \sum_{\langle i,j \rangle} s_i s_j - H \sum_i s_i$$

where the first sum is taken over all distinct nearest-neighbor pairs and the second sum is over all spins. H is an external magnetic field.

On an $L \times L$ square lattice, each spin has four nearest neighbors (left, right, up and down). Periodic boundary conditions are applied at the edges (these are explained in more detail soon).

In the following, we consider the Ising model in zero field $H = 0$. In zero field, this model has two degenerate ground states (states of minimum energy): either all spins up or all spins down. In the absence of an external field, these two states are equivalent.

At low temperatures, a typical state is close to the ground state with most spins being either up or down. When the temperature is raised, the system undergoes a **phase transition** which is characterized by the simultaneous occurrence of large segments of up and down spins. At high temperatures, the system is disordered and there are virtually no correlations between spins.

```
+ o o + o + + + + +
+ + + + + + o o o +
+ + o o o o o o o o
+ + + o o o o + + +
+ + + o o o + + + +
+ + + o o o + + + +
o + + o o o + + + o
+ + + + o + + + + +
+ + + + + + + + + +
o o + + o + + + + +
```

Figure 6: Example of a configuration of the 10×10 Ising model. '+' denotes the spin value of +1 and 'o' the value -1. Each spin interacts with its nearest neighbors only (for the 2D square lattice, each spin has four nearest neighbors.)

Figure 7 shows three snapshots of typical configurations obtained from a Metropolis Monte Carlo simulation of the Ising model in zero field. Black denotes up spins (+1) and white down spins (-1) or vice versa (for the zero field model, the two directions are equivalent).

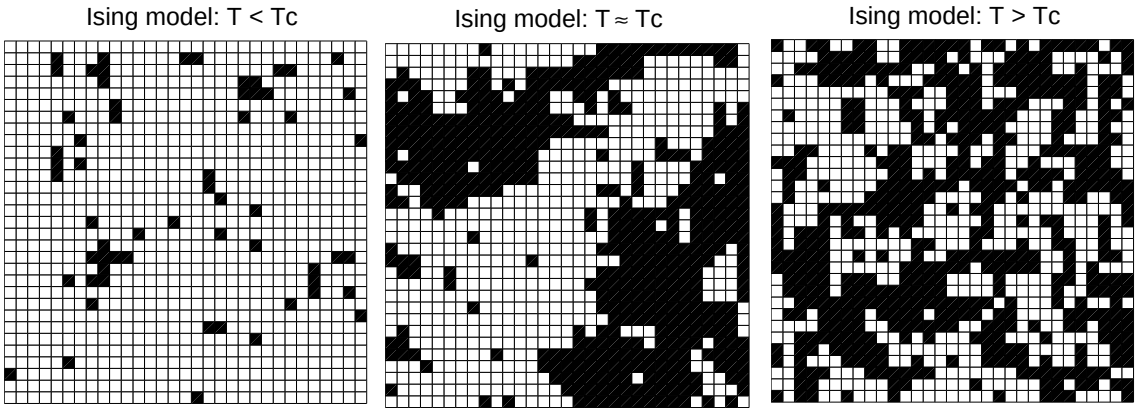


Figure 7: Typical spin configurations for the 2D Ising model.

3.2 Metropolis importance sampling

In the classic, Metropolis method configurations are generated from a previous state using a transition probability which depends on the energy difference between the initial and final states. The sequence of generated states follows a time ordered path, but the time is non-deterministic and referred to as 'Monte Carlo time'.

A **stochastic process** $S = \{r_i\}$ is a sequence of random variables r_i.

A *Markov chain* is a stochastic process for which

$$p(r_{i+1}|r_i r_{i-1} \ldots r_1) = p(r_{i+1}|r_i) \qquad \text{for all } i$$

This means that a Markov chain has no memory of the earlier states $i-1, i-2, \ldots$.

The variable $p_i(t) = p(r_i, t)$ is the probability that the system is in state i at time t. During a Markov process, the time-dependent behavior of the probabilities p_i is described by the

following Master equation

$$\frac{\partial p_i(t)}{\partial t} = -\sum_{j=1}^{N} [w_{ij} p_i - w_{ji} p_j]$$

where w_{ij} is the transition rate from state i to state j.

In *equilibrium*, there is a unique probability distribution $p*$ such that

$$\frac{\partial p*}{\partial t} = 0$$

This is guaranteed by choosing

$$w_{ij} p_i* = w_{ji} p_j*$$

This is called the *detailed balance condition.*

For a classical system in the **canonical ensemble**, the probability of the ith system occurring is given by

$$P_i = \frac{1}{Z} e^{-E_i/k_B T}$$

where Z is the partition function. This probability is usually not known because the partition function is practically impossible to calculate for most systems.

This difficulty can be avoided by using Markov chains: We can generate each new state directly from the previous state. This can be easily done because we only need to consider the relative probabilities of two successive states i and j. From the detailed balance condition we get

$$\frac{w_{ij}}{w_{ji}} = \frac{p_j}{p_i} = e^{-(E_j - E_i)/k_B T}$$

Thus we only need to know the energy difference $\Delta E = E_j - E_i$.

Transition rates

Any transition rate which satisfies the detailed balance condition is acceptable. The first and most widely used choice is the **Metropolis** form

$$w_{ij} = \begin{cases} \frac{1}{\tau} e^{-\Delta E/k_B T} & \text{if } \Delta E > 0 \\ \frac{1}{\tau} & \text{if } \Delta E \leq 0 \end{cases}$$

where τ is some (arbitrary) time scale (e.g. the time required to attempt a trial move). Often this time scale is set to unity.

An example of an alternative method is the **Kawasaki** or symmetric form

$$w_{ij} = \frac{1}{2\tau} [1 - \tanh(\Delta E/2k_B T)]$$

Metropolis importance sampling Monte Carlo algorithm

1. Generate an initial state i.
2. Attempt to change the configuration from i to j.
3. Compute $\Delta E = E_j - E_i$.
4. If $\Delta E \leq 0$, accept the change and replace state i by state j. Goto 2.
5. If $\Delta E > 0$, generate a uniform random number $r \in [0,1]$.
   Calculate $w = \exp(-\Delta E / k_B T)$.
6. If $r < w$, accept the change and replace state i by state j. Goto 2.
7. If $r > w$, then reject the change and keep state i. Goto 2.

This algorithm generates states according to the canonical distribution once the number of states is sufficiently large so that the system has reached equilibrium.

Calculation of average values

The average value of a given quantity A is given by the following sum

$$\langle A \rangle = \sum_i p_i A_i$$

The great idea behind the Metropolis algorithm is that once equilibrium has been reached, the average value of A is obtained simply as an arithmetic average over the entire sample of states generated in the simulation; i.e.

$$\langle A \rangle_{MC} = \frac{1}{N} \sum_{i=1}^{N} A_i$$

Note that if an attempted change is rejected, then the old state is calculated for averages again.

If the successive states are *uncorrelated*, then the estimate of the error is given by the standard deviation

$$\sigma = \sqrt{\frac{1}{N}(\langle A^2 \rangle - \langle A \rangle^2)}$$

In practice, consequent configurations are almost always strongly correlated (e.g. only a small local change can occur in each attempt and thus the consecutive states are very much alike). In cases where the correlation times are not very long, the easiest way to avoid the problem is to take every nth configuration (state) into the averages and discard the rest. If one wants to be sure that the samples are indeed uncorrelated, then a suitable value for n can be obtained by calculating the autocorrelation time τ_A for the system in question and setting $n >> \tau_A$.

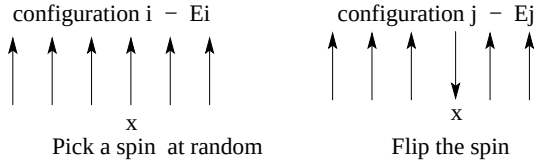
3.3 Updating schemes

In step 2 of the Metropolis algorithm, the configuration is changed from state i to state j. The algorithm does not state *how* this change should be made, only that the next configuration is somehow obtained from the previous one. The Metropolis algorithm is in fact a very general algorithm which can be applied to studying very different kinds of problems. In the following, we use the 2D Ising model as our example and introduce different types of moves that can be used in connection with the Metropolis algorithm to change the configuration. In the next lecture, we will see a different example where the Metropolis algorithm is applied to perform off-lattice simulations of semiconductor materials.

Updating schemes for the Ising model can be categorized into **local** and **global** schemes. In the former one, the trial configuration j is obtained from the current configuration i by doing a change which affects only a small group of spins in the system. Global schemes, on the other hand, aim at doing very large changes in the system at each step. The purpose is to speed up the calculation by generating a sequence of configurations which do not all belong to the same region of phase space. We consider two local updating schemes here.

Spin-flip update

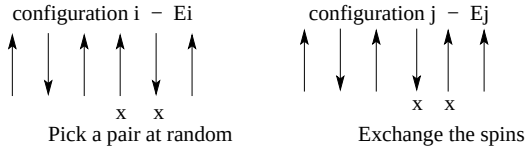
In the so-called spin-flip updating scheme, a new trial configuration is generated from the existing one by randomly picking a spin and flipping it (i.e. +1 becomes -1 and vice versa). We then proceed to calculate the resulting energy difference $\Delta E = E_j - E_i$ and use this to test whether the move is accepted or not. The energy difference is obtained by considering how the flip changes the energy of the spin in question and the energies of its four nearest neighbors.



This corresponds to *deposition / evaporation* of a particle in the lattice-gas picture (which means that we think that each lattice site is either occupied ($s_i = +1 \Rightarrow n_i = 1$) or not occupied ($s_i = -1 \Rightarrow n_i = 0$) by a particle. There is no conservation of magnetization (spin model) or particle number (lattice gas picture).

Spin-exchange update

In the spin-exchange update, we pick two neighboring spins at random and attempt to exchange the values of the spins. As in the spin-flip case, acceptance is tested by calculating the resulting energy difference $\Delta E = E_j - E_i$. In this case we need to consider the two spins and all their nearest neighbors.



This corresponds to *hopping* (or diffusion) of a particle in the lattice-gas picture. Magnetization / particle number is conserved.

3.4 Periodic boundary conditions

Since we always use a finite system in the simulations (e.g. an $L \times L$ lattice for the 2D Ising model), an important question is how to treat the edges of the lattice. In most cases, we are not interested in studying boundaries or surfaces. Therefore, an effective way to eliminate the boundaries is to 'wrap' the lattice to create a 'torus' structure. In the 2D case, this means that the first spin at the left edge of each row 'sees' the last spin of the row as its nearest neighbor and vice versa. The same applies for the top and bottom rows. This is termed the *periodic boundary conditions* (PBC).

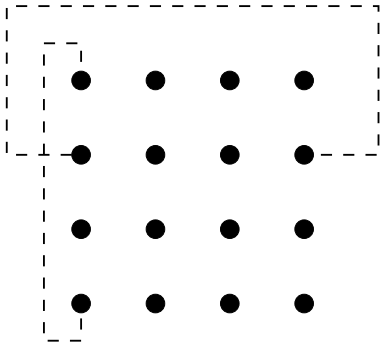


Figure 8: Periodic boundary conditions on a 2D square lattice.

3.5 Implementation of the Metropolis algorithm

In the following, we discuss how the Metropolis algorithm is implemented in practice to perform a simulation of the 2D Ising model.

First, we need an array of size $N = L \times L$ which can take the integer values ± 1. In practice, it is convenient to define this array as a pointer and then allocate memory once the size of the system is known. In C, the following segments of code can be used for defining the array, allocating memory and addressing the elements:

```
/* Macro for elements of 'lattice' */
#define R(x,y) (x + L*y)

/* Definition of the pointer */
int *lattice;

/* Memory allocation */
lattice = (int *) calloc((L*L),sizeof(int));
```

With these commands, an element (spin) of the lattice can be addressed by its x and y coordinates ($x, y \in [0, L-1]$):

```
spin = lattice[R(x,y)];
```

Periodic boundary conditions are applied to this lattice. Each spin has four nearest neighbors. For inner spins at position (x,y) (not at the boundary), the values of the neighboring spins are obtained from

```
nn[0] = lattice[R(x-1,y)]    /* left  */
nn[1] = lattice[R(x+1,y)]    /* right */
nn[2] = lattice[R(x,y+1)]    /* up    */
nn[3] = lattice[R(x,y-1)]    /* down  */
```

For boundary spins, the neighbors are obtained from

```
if(x=0)   nn[0] = lattice[R(L-1,y)]  /* left  */
if(x=L-1) nn[1] = lattice[R(0,y)]    /* right */
if(y=L-1) nn[2] = lattice[R(x,0)]    /* up    */
if(y=0)   nn[3] = lattice[R(x,L-1)]  /* down  */
```

There are various ways of implementing the PBC in practice (e.g. it is possible to create an initial list which allows a quick calculation of the neighboring spin values).

The next task is to decide the values of the temperature T and the interaction parameter J. The critical temperature of the Ising model (where the phase transition occurs) is approximately $k_{\mathrm{B}}T/J = 2.269 = T_c$. For simplicity we set $J = 1$ and express T in energy units (so that $k_{\mathrm{B}} = 1$). At $T = 0$, the system is in one of its two equivalent ground states (all spins up or down). At $T = \infty$, the thermal energy $k_{\mathrm{B}}T$ is infinitely larger than the energy due to the spin-spin interaction J, so the spins are oriented randomly up or down in an uncorrelated way.

The **initial state** of the simulation is commonly chosen to be either the ground state or the $T = \infty$ state. These are popular choices because both are easy to generate and correspond to a well-known temperature.

If we are performing a set of simulations at a range of different values of T, then a good choice for the initial state is to use the *final* state obtained from the previous simulation at a nearby temperature.

Note, however, that the outcome of the simulation must *not* depend on the initial state if the algorithm is working correctly. In practice, it might take much longer for the system to reach equilibrium if the simulation is started from an initial state which is very far from a typical equilibrium configuration. (We return to this question of equilibration in a moment.)

Now we are ready to **start the simulation**. The first step is to generate a new trial configuration using a predecided updating scheme; e.g. the spin-flip scheme. We pick a single spin from the lattice by selecting the x and y coordinates randomly from the interval $[0, L-1]$ (assuming that the indexing goes from 0 to $L-1$ as in C).

Then we calculate the energy difference ΔE which would result if the selected spin were overturned. A straightforward way to do this calculation is to first calculate the energy of the initial configuration from

$$E_0 = -J \sum_{\langle i,j \rangle} s_i s_j, \tag{1}$$

where $\sum_{\langle i,j \rangle}$ denotes summation over the nearest neighbours i and j. The sum over all nearest-neighbor pairs can be efficiently computed by sweeping through the lattice in an ordered way: Start from the position $x = 0$ and $y = 0$ and for each lattice site, calculate the contribution which comes from the *two* neighbors on the *right* and *above*. This way each pair is calculated once.

Then for each new trial state, we can calculate the **energy difference** $\Delta E = E_j - E_i$ by considering only *five spins* (the flipped spin and its nearest neighbors):

$$\Delta E = 2Js_i \sum_{k=1}^{4} s_k.$$

For the 2D Ising model, there are only a few different possibilities for the value of ΔE. These are obtained by considering all the different nearest-neighbor configurations. (The enumeration is left as an exercise). For computational efficiency, the corresponding transition rates can be calculated in the beginning of the simulation which saves a large number of evaluations of the exponential function.

After evaluating ΔE, we test for **acceptance** of the trial state. If $\Delta E \leq 0$, the trial state is accepted. Else, we draw a uniform random number $r \in [0,1]$. If $r < \exp(-\Delta E/T)$ ($k_B = 1$), the spin is flipped. Otherwise the move is rejected. Then the whole process is repeated for a new spin.

Time in local algorithms is measured in units of the average number of updates per spin. One **Monte Carlo step** (MCS) is defined as the time after which every spin in the system is attempted on average once.

For an $L \times L$ lattice, this means that we have a loop over Monte Carlo steps, and each step consists of $N = L \times L$ random spin-flip attempts. Many results are presented as a function of MCS.

Example code

The following segment of C code, shows the heart of the Metropolis Monte Carlo algorithm (periodic boundary conditions are applied using a predefined array 'pb'):

```
/* One MC step per spin with random spin flip dynamics */
/* ------------------------------------------------- */
void metropolis(int L, float *crn, float *RX, float *RY,
                int *pb,  int *nn1, int *nn2, double *E,
                double *M, double *wij, int *spin, int *acc)
{
  int i, x, y, dE, accept;

  /* Load random number arrays */
  cranmar(RX,L*L);
  cranmar(RY,L*L);
  cranmar(crn,L*L);

  for(i=0; i<L*L; i++) {

    /* Select lattice site randomly */
    x = pb[(int)(RX[i]*L)];
    y = pb[(int)(RY[i]*L)];

    /* Calculate energy difference */
    dE = deltaE(x,y,nn1,nn2,spin,L);

    /* Test for acceptance */
    if(dE <= 0)
      accept=1;
    else if(crn[i] <= wij[dE])
      accept=1;
    else
      accept=0;

    /* Update if accepted */
```

```
    if (accept==1) {
      /* Flip the spin */
      spin[R(x,y)] = -spin[R(x,y)];
      /* Magnetization */
      *M = *M + 2*spin[R(x,y)];
      /* Energy */
      *E = *E + (double)dE;
    }
  }
}
```

The energy difference is obtained from (the arrays 'nn1' and 'nn2' take care of the periodic boundary conditions):

```
int deltaE(int x, int y, int *nn1, int *nn2, int *spin, int L)
{
  int dE;

  dE = 2.0*spin[R(x,y)]*(spin[R(nn1[x],y)] + spin[R(nn2[x],y)]
                       + spin[R(x,nn1[y])] + spin[R(x,nn2[y])]);
  return(dE);
}
```

The transitions rates were initially tabulated using (note that only two(!) values are needed, why?):

```
  for(dE=4; dE<=8; dE+=4)
    wij[dE] = exp(-(double)dE/T);
```

3.6 Equilibration

How is the Metropolis Monte Carlo program for the Ising model then used in practice? Typical questions are for example: "What is the magnetization at a given temperature?" or "How does the internal energy behave in a given range of temperatures?".

In order to answer these questions, two things need to be done. First, we have to find out how long it takes for the system to reach *equilibrium*. Remember that we can start the simulation from any initial configuration (e.g. all spins randomly distributed), and the Metropolis algorithm will take the system into its equilibrium state. Equilibrium is characterized by average quantities remaining unaltered within statistical fluctuations. After we know the required equilibration time, we have to perform a simulation run which samples the equilibrium distribution for a sufficiently long time. During this run we measure the required average quantities.

We first discuss the question of equilibration. Figure 10 shows a series of three snapshots taken of a 32×32 Ising model at $T = 2.0 < T_c$. The initial configuration was random (corresponding to $T = \infty$). The first snapshot is taken after 1 MCS (after $N = 32 \times 32$ spin-flip attempts). Naturally, the spins are still very randomly distributed. In the second snapshot, after 10 MCS, we observe large coexisting segments of up and down spins. This means that spins like to be in the vicinity of the same type of spins. The white areas have begun to dominate and the black areas are shrinking. The last snapshot is after 10000 MCS which is well after the system has equilibrated. Here we see a typical equilibrium

configuration: the system is dominated by one type of spins and small sections of overturned spins are seen here and there (these will disappear if the temperature is lowered and the system approaches its ground state).

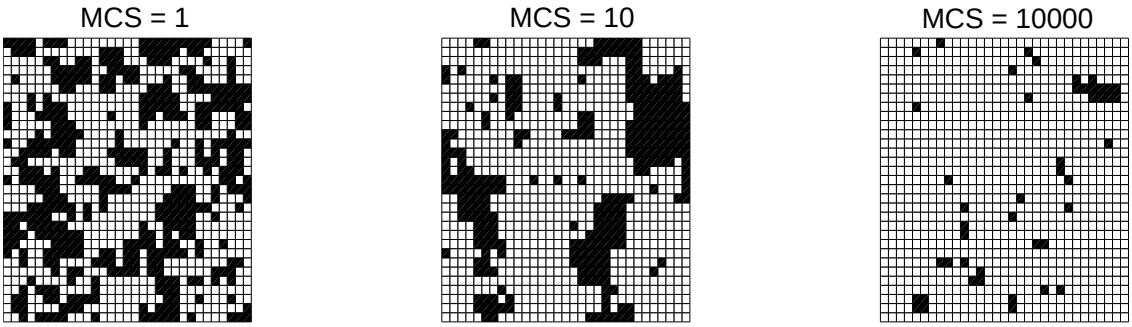


Figure 9: Snapshots of the lattice during equilibration.

Equilibrium means that the average probability of finding the system in any particular state i is proportional to the Boltzmann weight $\exp(-E_i/k_{\mathrm{B}}T)$ where E_i is the energy of state i.

If we start the simulation for example from a state which corresponds to $T = \infty$ (a frequently occurring or 'typical' state at this temperature), it will take some time before the system evolves through a sequence of states until equilibrium is reached. After this, the sequence of generated states has the correct Boltzmann distribution. The Metropolis algorithm is designed in such a way that it will first take the system to equilibrium and the system will stay in equilibrium for the rest of the simulation.

Looking at pictures is not an efficient way to determine if equilibrium has been reached. A better way is to plot a graph of some quantity of interest and see how this quantity behaves as a function of MC steps.

Magnetization per site is given by

$$m = \frac{1}{N}\sum_{i=1}^{N} s_i \qquad (N = L \times L)$$

where the sum is over all spins in the system.

Internal energy per site is given by

$$u = -\frac{J}{N}\sum_{\langle s_i, s_j \rangle} s_i s_j \qquad (N = L \times L)$$

where the sum is over all nearest neighbor pairs in the system.

Both quantities are calculated once for the initial state and updated after each accepted spin flip (not recalculated for the entire system).

Figure 10 shows the graphs of the magnetization and the internal energy per site as a function of time (in MCS).

The magnetization is initially close to zero corresponding to the random spin configuration and then starts to increase until a value close to 1 is reached. This corresponds to a state where most spins are up and only a small fraction of the spins point down. The fluctuations describe random changes in the system when the system samples the equilibrium

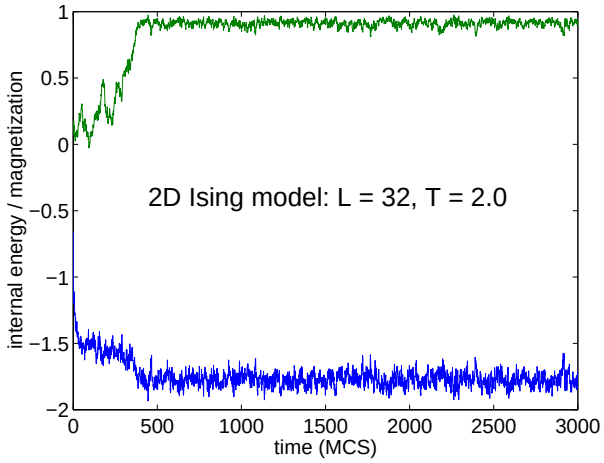


Figure 10: Magnetization and internal energy per site of the 2D Ising model on a square 32×32 lattice at $T = 2.0$. The simulation was started at a random configuration corresponding to $T = \infty$. Equilibrium is reached after about 500 MC steps (each step consist of $N = 32 \times 32$ updates).

distribution. Similarly, the internal energy decreases in the beginning until equilibrium is reached. After this, the values fluctuate around the same mean value.

Looking at these two graphs, we can say that equilibrium is reached after about 500 MC steps. After that, we could start measuring average values over the sequence of configurations generated as a function of MC steps (e.g. the mean magnetization $\langle m \rangle$ and the mean internal energy $\langle u \rangle$).

Independent runs

The problem of equilibration is, however, not solved yet. Looking at just one simulation run is not sufficient because different runs use a different sequence of random numbers which means that the systems will follow different paths to equilibrium. Consequently, one run may equilibrate much faster than another one does.

Therefore, we must plot graphs of some quantity of interest from at least *several independent runs*. Figure 11 shows a plot of two graphs obtained from two independent runs. We notice that the other run reaches equilibrium much faster, in about 200 MCS. If we had used this as our limit after which we begin measuring averages, then we would have obtained incorrect results from the second measurement (for which the run equilibrates only after about 500 MCS).

Different initial states

Another sensible thing to do is to perform some check runs using *different* initial states (*i.e.* states corresponding to different energies). This is because the system can get stuck in a *metastable* state which is a **local energy minimum**. The system can stay in this state temporarily and it may look like all the average quantities have reached constant values within fluctuations.

Such an incidence can usually be spotted by starting the simulation from different states. If everything works correctly, all runs should reach the same equilibrium state which is the **global energy minimum**. In a metastable state, most quantities will have different values than those in the true equilibrium.

Figure 12 shows graphs obtained from four different runs. All runs have different initial

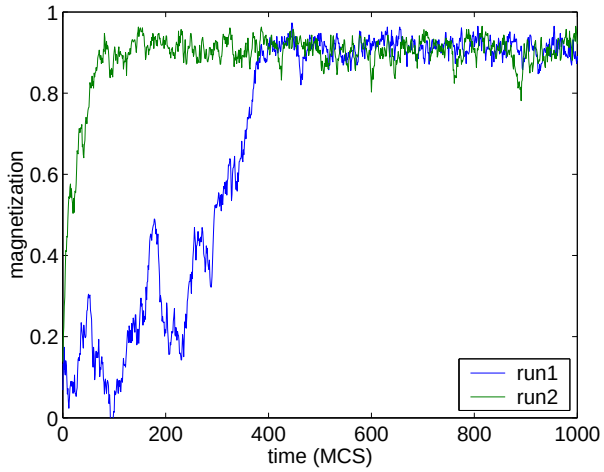


Figure 11: Magnetization of the 32×32 Ising model for two different simulations. The simulations were started in two different $T = \infty$ (random) configurations. At about 600 MC steps both runs have converged to the same mean value of magnetization, within statistical errors due to fluctuations. This means that both runs have equilibrated.

states and different random number sequences. In two cases, the magnetization converges to a value slightly below +1, while in the two other cases, the values converge to -1. Why is this?!

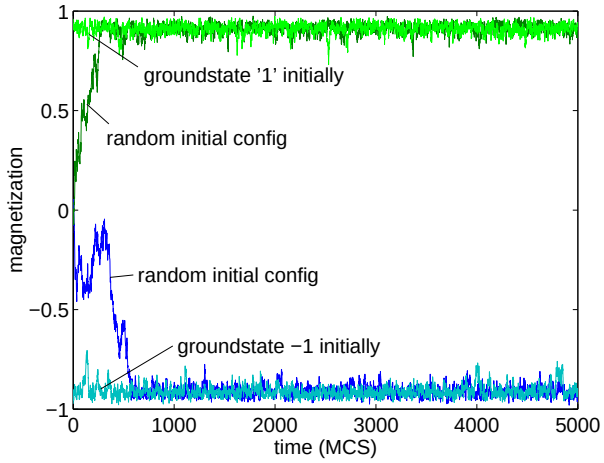


Figure 12: Magnetization and internal energy per site of the 2D Ising model on a square 32×32 lattice at $T = 2.0$. The simulation was started at a random configuration corresponding to $T = \infty$. Equilibrium is reached after about 500 MC steps (each step consist of $N = 32 \times 32$ updates).

Double-peaked distribution of the Ising model below T_c

The reason for the convergence to different values of the magnetization is explained by the existence of two degenerate ground-states (all spins up or all spins down). At temperatures below T_c, this is reflected by a highly double-peaked probability distribution $P(M)$ of the total magnetization M (or equivalently of the magnetization per site, m). Figure 13 shows the magnetization distribution $P(M)$ of the 32×32 Ising model at $T = 2.2$ obtained from a simulation with 200000 measurements.

The double-peaked distribution implies that the states between the peaks (which have comparable amounts of both up and down spins) are hardly ever sampled. This explains

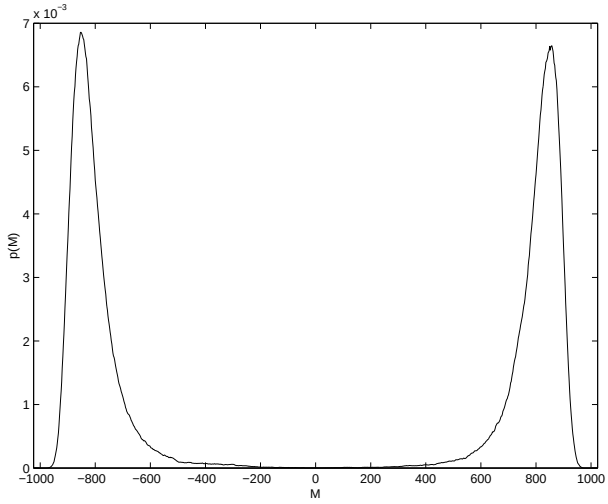


Figure 13: Probability distribution $P(M)$ of magnetization M. Note that this is difficult to measure using standard MC techniques because the intermediate states occur very rarely. 200000 measurements were needed, although the lattice size is only 32×32.

why different simulation runs seem to converge to different values of the magnetization: The runs are sampling states around one of the peaks. If we were to continue the runs for a much longer time, then eventually we would see switches of magnetization (the system traversing from one peak to the other).

3.7 Measurements

Once equilibrium is reached, we can begin to measure whatever quantities we are interested in. Typical quantities are the magnetization m and internal energy u per spin.

The total internal energy E is updated during the simulation after each accepted spin flip by

$$E_{new} = E_{old} + \Delta E$$

where ΔE is the energy change which has been calculated in the Metropolis algorithm in order to test for acceptance.

Similarly, the magnetization changes in each accepted spin flip by

$$\Delta M = 2s_i$$

where s_i is the value of the flipped spin after the flip has taken place.

We can measure the average values of these quantities during a simulation run by simply calculating the average obtained over sufficiently many MC steps. However, the configurations obtained from successive MC steps are strongly correlated. A truly satisfactory solution would be to measure the *correlation time* of the simulation. This measures how long it takes for the system to get from one state to another which is significantly different from the first.

We will not go into such detail here but take a simple approximative way to solve the problem: We look at the graph of the measured quantity and decide on some suitable interval to wait before updating the averages. In the case of the 32×32 Ising model at $T = 2.0$, taking data every 100 MCS will most likely be sufficient.

Then taking a total of n measurements at the decided intervals, we can use the following formula for calculating an estimate of the error; e.g. for the magnetization

$$\sigma_m = \sqrt{\frac{1}{n-1}(\langle m^2 \rangle - \langle m \rangle^2)}$$

Here is an example result obtained from a single simulation run:

```
                2D Ising model
--------------------------------------------------
Spin-flip dynamics with the Metropolis algorithm
Input parameters:
System size: LxL = 32x32
Number of sweeps: N = 40000
Temperature: T = 2.000000
Coupling constant: J = 1.000000
Seed for RNG: seed = 83844

 DATA TAKEN EVERY 200 MCS:

 <u>   = -1.736836
 err   = 0.002791

 <|m|> = 0.909546
 err   = 0.001494
```