universität innsbruck
Institut für Theoretische Physik

Prof. Dr. T. Franosch

Dr. Michele Caraglio

Institut für Theoretische Physik

WiSe 2021 5.11.2021

Computer Simulations in Statistical Physics

Prof. Dr. T. Franosch

Problem set 3

Proseminar

Problem 3.1 *Ising model*

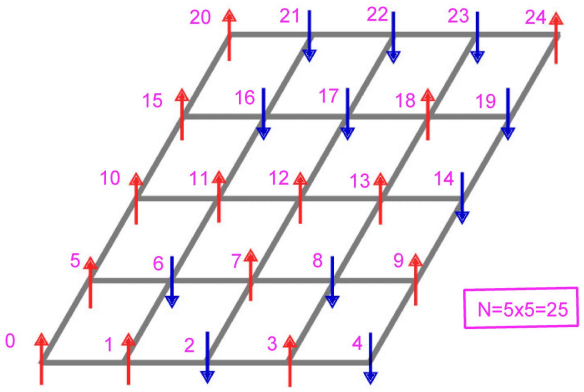
The Ising model is a cornerstone of classical statistical mechanics. It is a mathematical model of ferromagnetism that can be solved exactly in one and two dimensions. In two and higher dimensions the model shows a phase transition.
The Ising model it is defined on a lattice in d dimensions, whose sites are occupied by a spin taking two values $s = \pm 1$. The energy of a configuration $\{s_k\}$ is given by

$$E\left(\{s_k\}\right) = -J \sum_{\langle i,j \rangle} s_i s_j \ , \tag{1}$$

where $J > 0$ is a coupling constant and the sum is extended to nearest neighbors. In the following, we set $J = 1$.

a) Which is the energy of a completely ordered configuration with all spin values equal to 1? What do you expect for the energy of a random configuration?

b) Generate a random configuration of N spins displaced over a square lattice with periodic boundary conditions (take $N = 10 \times 10$) and compute its energy. Programming Hint: think object oriented, i.e. define a class lattice with its own relevant functions. When you add a function to a class, allways test if it is working!
Also remember: it is faster to deal with arrays of size N than with a matrix $\sqrt{N} \times \sqrt{N}$!



c) Repeat the previous point for $M = 100$ different random configurations. Plot the energy E_k with $k = 1, \ldots, M$ and compare the result with the energy of a configuration in which all the spins are equal to 1.

d) use the code snippet "*plot_Ising_configuration.py*" to visualize the configurations.

Problem 3.2 *Ising model - Metropolis algorithm*

We discuss here the Metropolis algorithm for the Ising model. In this algorithm spin configurations are sampled by a Markov chain, which is characterized by a transition probability $P(\mu \to \nu)$ where μ and ν are two spin configurations. In the algorithm $P(\mu \to \nu) \neq 0$ only if the two configurations differ by a single spin. We initialize the system by selecting a random configuration μ of a $N = L \times L$ lattice with periodic boundary conditions. The practical implementation of the algorithm is as follows:

1. Select one of the N spins at random, say spin s_i.
2. Calculate $\Delta E = E_{fin} - E_{in}$, where E_{in} is the energy of the present configuration and E_{fin} the energy of the configuration obtained by flipping the spin $s_i \to -s_i$.
3. Flip the spin $s_i \to -s_i$ if $\Delta E \leq 0$ or if $r \leq \exp(-\Delta E/k_B T)$, where r is a uniform random number in $[0, 1]$.
4. Go to (1).

Typical observables one computes in the Ising model are the energy, as defined by Eq. (1), or the magnetization per spin defined as

$$m = \frac{1}{N} \sum_{i=1}^{N} s_i \ , \tag{2}$$

or spin-spin autocorrelations functions. These quantities reach their equilibrium value after some equilibration time τ_{eq}. Another quantity we compute is the (temporal) autocorrelation function which is defined for a system in which time is a discrete variable

$$\chi(t) = \frac{1}{T} \sum_{s=\tau_{eq}}^{\tau_{eq}+T} \left[m(s) - \langle m \rangle\right] \left[m(s+t) - \langle m \rangle\right] \ , \tag{3}$$

where $m(t)$ is the magnetization at timestep t, and $\langle m \rangle$ is the average equilibrium magnetization. This equation is only valid when the Monte Carlo run has reached an equilibrium configuration ($t > \tau_{eq}$). This autocorrelation function decreases exponentially like

$$\chi(t) \sim \exp\left(-t/\tau\right) \ , \tag{4}$$

where τ defines the autocorrelation time of the simulation. This is the timescale of the simulation, and it indicates how much time it takes to arrive at a new uncorrelated configuration of the model.

a) For a temperature $T = 2$ and $N = 2500$ and starting from a random configuration, use the Metropolis algorithm to compute and plot the energy and the magnetization per spin as a function of the Monte Carlo time step. (For making the plot use the code snippet *"Plot_Ising_trajectory.py"*) Compare the average magnetization with the exact value (valid in the ferromagnetic phase):

$$m(T) = \left[1 - \sinh^{-4}\left(2J/T\right)\right]^{1/8} \ , \tag{5}$$

b) From your plot estimate the equilibration time τ_{eq}.

c) Use the code snippet *"Plot_Ising_trajectory.py"* to visualize time evolution.

d) Determine the correlation time τ (For making the fit and plotting use the code snippet *"Plot_Ising_trajectory.py"*)

universität innsbruck
Institut für
Theoretische Physik

Prof. Dr. T. Franosch

Dr. Michele Caraglio

Institut für Theoretische Physik

WiSe 2021 5.11.2021

Computer Simulations in Statistical Physics

Prof. Dr. T. Franosch

Problem set 3

Solution to Problem 3.1 *Ising model*

We start by defining a class "lattice". In this class are contained all relevant functions and subroutines that we can implement on a lattice configurations.

file = lattice.h

```
class Lattice {
  public:
    Lattice(int, double [], double []);      // Constructor
    int get_spin_index(int, int);
    double get_spin_value(double [], int, int);
    int get_right(int);
    int get_left(int);
    int get_above(int);
    int get_below(int);
    double get_energy(double []);

  private:
    int L;
};
```

file = lattice.cpp

```
#include <math.h>
#include "lattice.h"
#include "nrutil.h"

Lattice::Lattice(int N, double s[], double s_initial_values[]){
  // call constructur make a lattice with periodic boundary,
```

```
  // spins initially setted equal to the array "s_initial_values"
  L = sqrt(N);
  for (int i =0; i<N; i++){
    s[i] = s_initial_values[i];
  }
}

int Lattice::get_spin_index(int i, int j){
  // i (j) = row (column) index from 0 to L-1
  // Output: the spin index corresponding to (i,j)
  int k = i*L + j;
  return k;
}

double Lattice::get_spin_value(double s[], int i, int j){
  // i (j) = row (column) index from 0 to L-1
  // Output: the spin value corresponding to (i,j)
  int k = i*L + j;
  return s[k];
}

int Lattice::get_left(int i){
  // return the index of the site at the left of site i
  int j=-1;
  if (i%L == 0){
    j = i+L-1;
  } else {
    j = i-1;
  }
  return j;
}

int Lattice::get_right(int i){
  // return the index of the site at the right of site i
  int j=-1;
  if ((i+1)%L == 0){
    j = i+1-L;
  } else {
    j = i+1;
  }
  return j;
}

int Lattice::get_below(int i){
  // return the index of the site above of site i
  int j=-1;
  if (i<L){
    j = i + (L-1)*L;
  } else {
    j = i-L;
  }
  return j;
}
```

```
60
61
62 int Lattice::get_above(int i){
63   // return the index of the site below of site i
64   int j=-1;
65   if (i>=L*(L-1)){
66     j = (i+L) % (L*L);
67   } else {
68     j = i+L;
69   }
70   return j;
71 }
72
73 double Lattice::get_energy(double s[]){
74   // return the energy E of a given configuration
75   int N = L*L;
76   double E = 0.0;
77   for (int i =0; i<N; i++){
78     E -= s[i]*s[get_right(i)] + s[i]*s[get_below(i)];
79   }
80   return E;
81 }
```

Then we write a very compact "main" file.

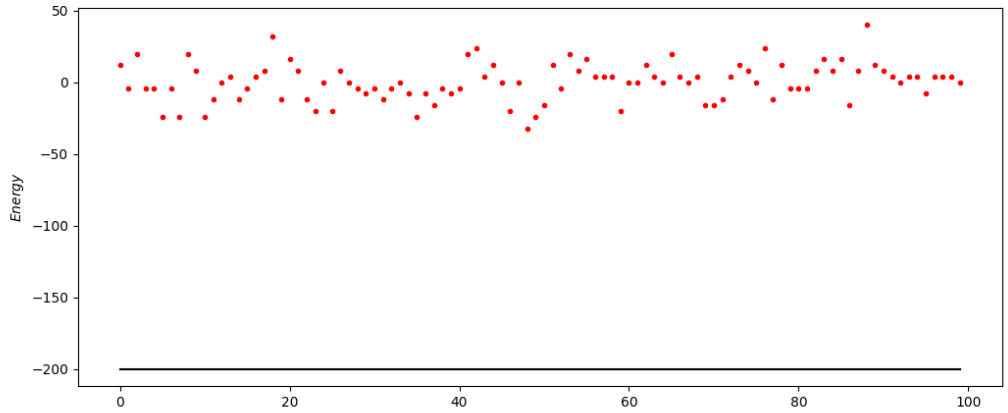
file = Ising.h

```
1  int L;
2  int N;
3
4  double * s;
5  double * s_initial_values;
6  double * s0;
7  double * s1;
8  double * s2;
9  double * s3;
10 double * s4;
11 double * s5;
12
13 double * mvector;
14 double mean (double x[], int NN);
15 double variance (double x[], int NN);
16 double autocorrelation (double x[], int NN, int tau);
```

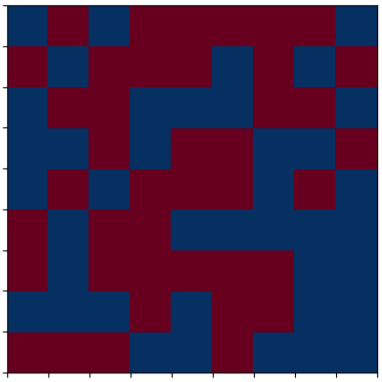
file = Ising.cpp

```
1 // compile with: g++ Ising.cpp lattice.cpp nrutil.cpp -o test -O2 -w
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <math.h>
5 #include <sstream>
6 #include "Ising.h"
7 #include "lattice.h"
8 #include "nrutil.h"
9
```

```
int main (){
  // set parameters
  L = 10;
  N = L*L;

  // array of length N
  s = dvector(0,N-1);

  // array for a completely ordered configuration
  s_initial_values = dvector(0,N-1);
  for (int i =0; i<N; i++){
     s_initial_values[i] = 1.0;
  }

  /* define lattice as an element of the Lattice class in the
     initial configuration defined by the array "s_initial_values" */
  class Lattice *lattice;
  lattice = new Lattice(N, s, s_initial_values);
  // get the energy of this configuration
  double Energy_order = lattice->get_energy(s);
  printf("%f \n",Energy_order);

  // open output file
  FILE * out;
  out = fopen("output1.dat","w");

  // random configurations
  for (int j=0; j<100; j++){
    for (int i =0; i<N; i++){
      double x = rand()/((double) RAND_MAX)-0.5;
      if (x>0) {
        s[i]=1.0;
      } else {
        s[i]=-1.0;
      }
    }
    double Energy = lattice->get_energy(s);
    fprintf(out,"%d %.4f %.4f \n",j,Energy_order,Energy);
  }
  fclose(out);

  out = fopen("output2.dat","w");
  for (int i =0; i<N; i++){
    fprintf(out,"%.1f \n",s[i]);
  }
  fclose(out);

  return 0;
}
```



Energy of different realizations of random configuration (red points) vs. the energy of a completely ordered configuration (black line).



Visualization of a random configuration.

Solution to Problem 3.2 *Ising model - Metropolis algorithm*

Add the following modules to the lattice class:

file = lattice.cpp

```
double Lattice::get_magnetization(double s[]){
  // return the magnetization m of a given configuration
  int N = L*L;
  double m =0.0;
  for (int i =0; i<N; i++){
    m += s[i];
  }
  m = fabs(m)/N;
  return m;
}
```

```
11
12 void Lattice::Metropolis (double s[], double T, int nmcstep){
13   // perform nmcstep Monte Carlo time steps
14   // using the Metropolis Algorithm
15   // T = temperature
16   int N = L*L;
17   double beta = 1./T;
18   double * precalc;
19   precalc = dvector(-4,4);
20   for (int i=-4; i<=4; i++){
21     precalc[i] = exp(-beta*2*i);
22   }
23   for (int t=0; t<nmcstep; t++){
24     int i = rand() % N;
25     double sum_neigbours = s[get_below(i)] + s[get_above(i)] +
26                            s[get_right(i)] + s[get_left(i)];
27     double E_in = -s[i] * sum_neigbours;
28     double E_fin = s[i] * sum_neigbours;
29     int j = int((E_fin-E_in)/2);
30     if (j<=0) {
31       s[i] = -s[i];
32     } else {
33       if (precalc[j] > rand()/((double) RAND_MAX)) {
34         s[i] = -s[i];
35       }
36     }
37   }
38 }
```

Obviously also the *lattice.h* should be modified accordingly.

And wrap up a main file:

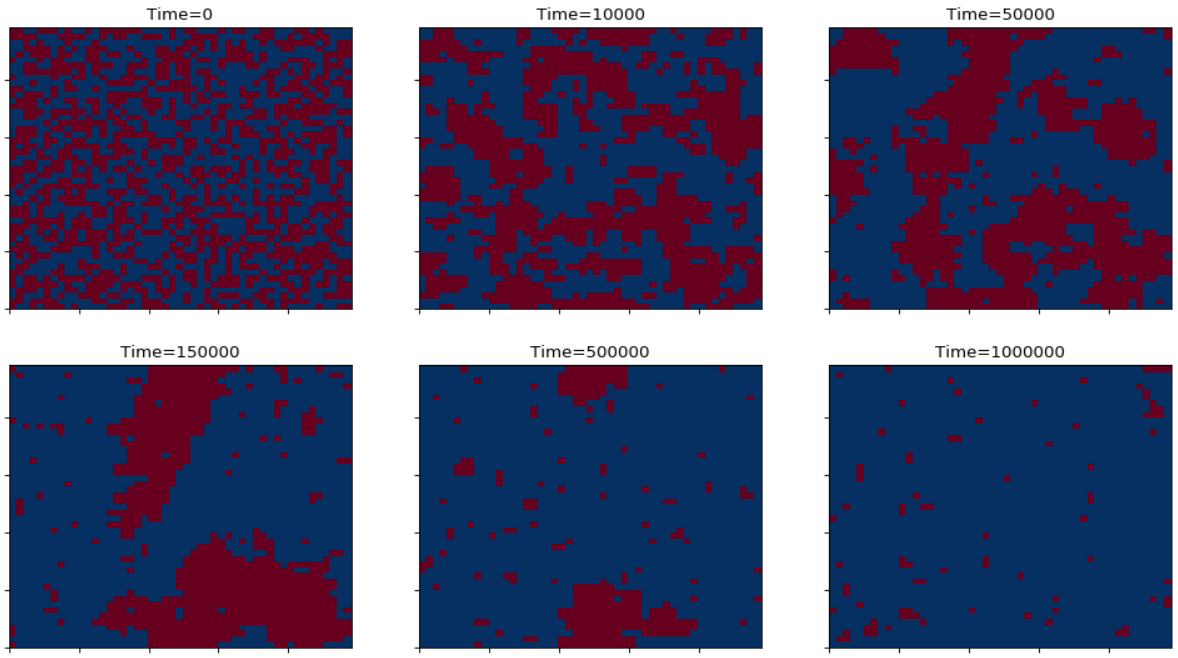
file = Ising_Metropolis.cpp

```
1 // compile with:
2 //   g++ Ising_Metropolis.cpp lattice.cpp nrutil.cpp -o test -O2 -w
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <math.h>
6 #include <sstream>
7 #include "Ising.h"
8 #include "lattice.h"
9 #include "nrutil.h"
10
11 int main (){
12   // set parameters
13   L = 50;
14   N = L*L;
15   double T = 2.0;
16   int nmcstep = 10000;
17
18   printf("critical temperature = %.4f \n", 2./log(1.+sqrt(2.)));
19   double ma = pow((1.- pow(sinh(2./T),-4.)), 0.125);
```

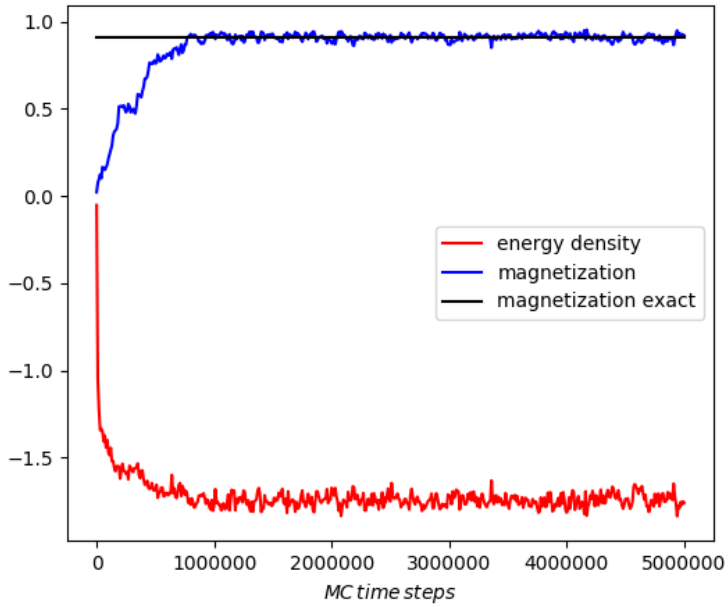
```
20
21    // array of length N
22    s = dvector(0,N-1);
23    s_initial_values = dvector(0,N-1);
24    s0 = dvector(0,N-1);
25    s1 = dvector(0,N-1);
26    s2 = dvector(0,N-1);
27    s3 = dvector(0,N-1);
28    s4 = dvector(0,N-1);
29    s5 = dvector(0,N-1);
30
31    // random configuration
32    for (int i =0; i<N; i++){
33      double x = rand()/((double) RAND_MAX)-0.5;
34      if (x>0) {
35        s_initial_values[i]=1.0;
36      } else {
37        s_initial_values[i]=-1.0;
38      }
39    }
40    class Lattice *lattice;
41    lattice = new Lattice(N, s, s_initial_values);
42
43    // open output file
44    FILE * out;
45    out = fopen("output3.dat","w");
46    double m =  lattice->get_magnetization(s);
47    fprintf(out,"%d %.1f %.4f %.4f \n",0,lattice->get_energy(s), m, ma);
48
49    for (int k =0; k<N; k++) {s0[k]=s[k];}
50
51    // Monte Carlo - Metropolis rule - equilibration
52    int l=0;
53    int j=1;
54    int i=0;
55    while (i<3*j && l==1 || l==0){
56      i++;
57      lattice->Metropolis (s, T, nmcstep);
58      m =  lattice->get_magnetization(s);
59      fprintf(out,"%d %.1f %.4f %.4f \n",
60                           (i+1)*nmcstep,lattice->get_energy(s), m, ma);
61      if (i==1) {
62        for (int k =0; k<N; k++) {s1[k]=s[k];}
63      } else if (i==5) {
64        for (int k =0; k<N; k++) {s2[k]=s[k];}
65      } else if (i==15) {
66        for (int k =0; k<N; k++) {s3[k]=s[k];}
67      } else if (i==50) {
68        for (int k =0; k<N; k++) {s4[k]=s[k];}
69      } else if (i==100) {
70        for (int k =0; k<N; k++) {s5[k]=s[k];}
71      }
72
```

```
    if (m>ma && l==0){
      printf("relaxation time = %d \n", (i+1)*nmcstep);
      j=i;
      l=1;
    }
  }
  fclose(out);

  out = fopen("output4.dat","w");
  for (int i =0; i<N; i++) {
    fprintf(out,"%.1f %.1f %.1f %.1f %.1f %.1f\n",
                        s0[i],s1[i],s2[i],s3[i],s4[i],s5[i]);
  }
  fclose(out);

  // Magnetization autocorrelation function
  //~ nmcstep=1000;
  //~ int nsave = 500000;
  //~ mvector = dvector(0,nsave-1);
  //~ for (int k=0; k<nsave; k++){
    //~ lattice->Metropolis (s, T, nmcstep);
    //~ mvector[k] =  lattice->get_magnetization(s);
  //~ }
  //~ out = fopen("output5.dat","w");
  //~ for (int k =0; k<nsave-10; k++){
    //~ fprintf(out,"%d %f\n",k*nmcstep, autocorrelation (mvector,nsave,k));
  //~ }
  //~ fclose(out);


  return 0;
}

double mean (double x[],int NN){
  // define the mean function
  double xm=0.0;
  for (int i =0; i<NN; i++){
    xm += x[i];
  }
  xm = xm/NN;
  return xm;
}

double variance (double x[],int NN){
  // define the variance function
  double xm=0.0;
  for (int i =0; i<NN; i++){
    xm += x[i]*x[i];
  }
  xm = xm/NN - mean(x,NN)*mean(x,NN);
  return xm;
}

```

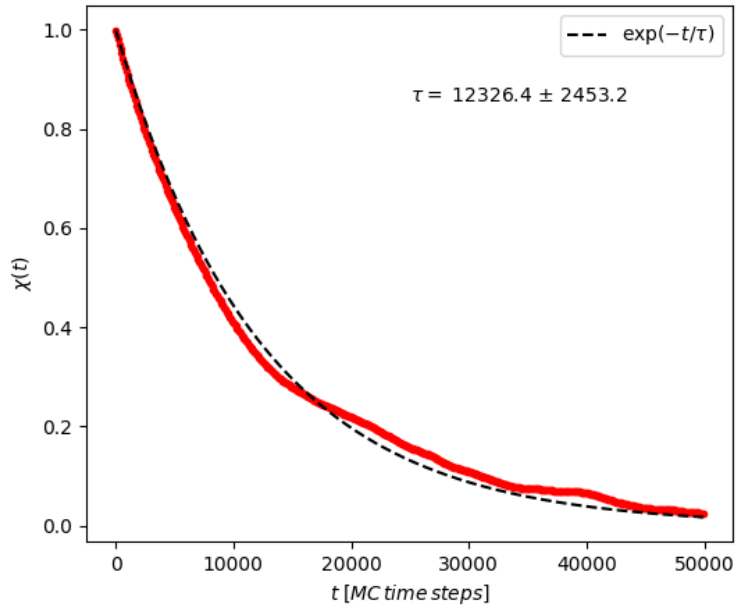
```
126 double autocorrelation (double x[], int NN, int tau){
127   // define the autocorrelation function
128   // input:
129   //   x = 1D array of data
130   //   tau = time lapse
131   // output:
132   //   c_n = autocorrelation of xs at time lapse n
133   double mu1 = mean (x,NN);
134   double sigma2 = variance (x,NN);
135   double c = 0.0;
136   for (int i =0; i<NN-tau; i++){
137     c+=(x[i]-mu1)*(x[i+tau]-mu1);
138   }
139   c = c/(NN-tau);
140   c = c/sigma2;
141   return c;
142 }
```



Visualization of the evolution starting with a random configuration.



Magnetization and energy density as a function of time.



Autocorralation function with a fit to the exponential decay.

Some interesting tricks: A Monte Carlo code consists of a central core nucleus which is repeated a large number of times. Therefore we should try to speed up this core to have a very efficient program (with a fast program we can sample a large number of different configurations

and thus get an accurate result).

- A common mistake is to compute the exponentials at each Metropolis step. The $\exp(x)$ is an operation which requires a certain amount of polynomial approximation on a computer and it is very slow. Since in the Ising model there are only few discrete values for $\Delta E > 0$ (how many?). It is much faster to calculate these once and store these numbers in a small vector.

- To calculate ΔE we only need to know the 4 spins up, down, right and left of the given spin, because all the other spins are unchanged.

- Often one performs simulations at different temperatures, say $T_1 < T_2 < T_3 \ldots$. To avoid long runs to reach equilibration one can take as begin configuration for the simulation at T_2 the last spin configuration of the run at T_1, and so on.