Computational Physics Report for Project B

Jianhao Wu

2022 fall Term

Abstract

This is the report for Project B of PHYS4061. In project B, students are required to complete a project about Monte Carlo method.

The problem I chose is 2D ising model. I have used two methods, single-spin flip and Wolff Algorithm to solve the problem, latter of which would perform better.

In this report, I would introduce some background knowledge firstly in the section 1 about our problem: ising model, and method: Monte Carlo Method. Next, in section 2 and section 3, I would introduce the principle, implement and result about these two methods. I would also introduce the concept of metastable configuration in subsubsection 2.3.3, which is the weakness of SSF algorithm and also one of our motivations to carry out Cluster Flip method. Finally I made a short summary in section 4.

Contents

1 Introduction

1.1 MC: Monte Carlo method

Monte Carlo is a city in Monaco which is famous for its gaming industry. This city's name was used by Neumann and Ulam [2] to represent the sampling method they concluded.

The main idea of Monte Carlo method is quite simple, just using random number generator to

sample randomly from a system. And if we could ensure the samples are "typical"(it means two aspects, the sample number needs to be large enough and the sample process should not be biased from the real system) and we are interested in some physical properties of this system, then we could calculate these of the samples as an approximation for the whole system.

1.1.1 Importance Sampling for normal case

Importance Sampling means we may give different weights to different candidates. It is an commonly used improvement method in Monte Carlo method.

Let us give an example of normal importance sampling:

Suppose there is a master's program which has 10 international students and 100 local students. And I want to make 10 questionnaires to study the workload of this master's program. Then if I distributed the questionnaires randomly, it is likely that no international students who have been asked. Thus the reliability of the result would be affected.

So I just choose to give 5 to the 10 international students and 5 to the 100 local students. But a new problem has come out: if I just take average of the result, it would enlarge the opinions of the international students, which is also not good. So I would take a weight 0.1 for every international student's result, and weight 1 for every local student, then the total weight between two groups observes the population relationship:

$$(0.1 * 5) : (1 * 5) = 10 : 100 \tag{1}$$

So the result now could not only ensure not omitting the opinions of the smaller group, but also promise the opinion's weight is equal to the population.

1.1.2 Importance Sampling by partition function

In physics, especially statistically physics, we know the probability of the appearing of micro state is proportional to its weight in the partition function of the system:

$$P(E_i) \propto e^{-\beta E_i} \tag{2}$$

where

$$\beta = \frac{1}{K_b T} \tag{3}$$

, K_b is the Boltzmann constant, i represents the ith micro state.

And the partition function of the system is given by:

$$\sum_i e^{-\beta E_i} \tag{4}$$

Since we mentioned above, i represents different micro states, so different state could have the same energy, resulting in the degeneracy in energy levels. Thus if we sum over different energy levels, then the partition function could be written as:

$$\sum_E \Omega(E) e^{-\beta E} \tag{5}$$

where $\Omega(E)$ is the degeneracy for the energy level E.

1.1.3 MCMC: Markov Chain Monte Carlo

As we have mentioned in 1.1.2, in physics we usually need to sample with the importance given by partition function. But how to do it?

To achieve this goal, we could combine Markov Chain and Monte Carlo.

Markov Chain has such a property: if a Markov Chain process has the below two conditions, then the micro state's probabilities under Markov Chain's steady state(after large enough steps) would follow a certain probability.

Ergodicity Any two states could reach each other.(Their probabilities between each other are higher than 0)

Detailed Balance The total probability from state a to b, is equal to the probability from b to a. Here "Total Probability" means we need to not only consider the transformation probability between two micro states but also the micro state's probability under the steady form.

$$P_\mu \cdot P_{\mu\to\nu} = P_\nu \cdot P_{\nu\to\mu} \tag{6}$$

1.2 Ising Model

Ising model is a quite simplified, but can be used to represent ferromagnetism, physics model with the following features:

1. Ising model was composed of plenty of atoms, which are arranged periodically. In 1D, 2D, 3D spaces, the atoms are accordingly arranged in a line with same distance, in a square, in a cubic.

2. We only consider one property for each atom: spin. The spin could be +1 or -1. So firstly, at least the mutual interactions between atoms would contribute to the total energy.

3. External magnetic field could also be considered, thus secondly, the total energy may be affected by the external field.

4. Combining the above two points, we could construct the Hamiltonian for Ising model:

$$H = -J \sum_{i,j} \sigma_i \sigma_j - \mu H \sum_k \sigma_k \tag{7}$$

5. One thing to note! In ising model, we have applied a simplification: only considering the interactions between the closest atoms, so the Hamiltonian could be written as:

$$H = -J \sum_{i \text{ and } j \text{ are the 1st nearest neighbor}} \sigma_i \sigma_j - \mu H \sum_k \sigma_k \tag{8}$$

1.2.1 2D square ising model: the simplest statistical models to show a phase transition [1]

1D ising model has been given a analytical solution by Ising himself in 1924, showing there is no phase transition [3].

So the model we chose is the 2D ising model, which is characterized by square lattice, which could show a phase transition.

What is expected to see for phase transition Phase transition means around the critical temperature, some important physical quantities would change suddenly, like magnetic susceptibility. So we could assume the middle point of the sudden change's range as the critical temperature.

Some further simplifications To simplify it further, we usually choose the case that H=0 and J=1, so then the Hamiltonian we consider becomes:

$$H = - \sum_{i \ and \ j \ are \ the \ 1st \ nearest \ neighbor} \sigma_i \sigma_j \tag{9}$$

1.3 Organization of my code files

Let me give an introduction to the files attached with my report. They are divided into three directories: Core Code, Python Code for drawing, and Data. The first one is the main c++ codes for Single Spin Flip and Wolff Algorithm, the second one includes some python files which could draw different pictures from the result generated by the first one. The third one is the data I have generated. **Besides, an directory named *Project B additional data* is also attached, which contains the raw data of the Wolff examples.** You could just use ConfigurationInMatplotlib.py in the Python Code for drawing directory to generate the certain number configuration images.

1. Core Code

   canvas.h & canvas.cpp The major part of both SSF algorithm and Wolff algorithm. The head file introduced the role of every functions in the cpp file.

   main.cpp Users could use this file to carry out SSF and Wolff by adjusting the parameters in the main.cpp.

2. Python Code for drawing

   ConfigurationPlot.py This py file could help to print the configuration image for one configuration file. It is used for quick looking at one configuration.

   ConfigurationInMatplotlib.py This py file could help to print the configuration image for many configuration files. If you want to generate many images in loop, you should use this one instead of the above one.

   PlotForProjectB.py This py could be used to plot 3D image for SSF result file, whose x coordinate is step number, and y coordinate is temperature.

   PlotForWolff.py This py has the same effect as PlotForProjectB.py, but for Wolff algorithm.

3. Data

SSF It has two directories, Pos and Rand, containing the data for these two kinds of initialization ways. Besides, an excel named SSF.xlsx has all the organized result of SSF experiment and all the figures I used in this report.

Wolff It has three directories, 300, 1000 and 2000, containing the data for my try on these three parameter options. Besides, an excel named Wolff.xlsx has all the organized result of Wolff experiment and all the figures I used in this report.

Let me show my head file, which has detailed explanations in the comment part thus could introduce the core functions in my cpp file:

Listing 1: **canvas.h**

```
#pragma once
#include <string>
using namespace std;
const double Kb = 1.380649E-23; //m^2 kg s^-2 K^-1

class canvas
{
public:
    //Initialization
    canvas(int size, double J, double H, double KbT, int PorNorR);
    //Deleting grid, since others could be deleted automatically
    ~canvas();

    //The next 4 are initialized in the Generate function
    //which has included the calculate functions(for all and average)
    //It means in the initial state we have updated them
    double TotalEnergy;//total energy, here the formula = -J\sum \
       sigma_i*\sigma_j-H\sum \sigma_k
    double TotalM;//=\sum \sigma
    double AverageEnergy;
    double AverageM;
```

```
21
22      int SingleStep=0;//Record how many steps for Single-Flip now
23
24      int GenerationWay;//0 for random, 1 for all positive, -1 for all
            negative
25      void RandomGenerate();
26      void AllPositiveGenerate();
27      void AllNegativeGenerate();
28
29      //It is only a simple version print configuration function in C++,
30      //for more beuautiful picture the python files could be used
31      bool PrintConfiguration();
32
33      //Save the configuration to a txt file
34      bool SaveConfiguration(string TxtName);
35
36      //Do Single-Spin-Flip for one time
37      //The decision of which site is flipped, is not included in this
            function
38      bool SingleSpinFlip(int row, int column);
39
40      //Currently only for the case H=0
41      //Return the cluster size in this flip activity
42      int WolffFlip(int row, int column);
43
44      //In SSF and Wolff, we only calculate the total physical quantities
45      //because calculate the average for every time would cause more time
            , and it's also unnecessary
46      void CalculateAverage();
47  private:
48      //The next variables are initialized in the constructor function
49      int** grid;
50      int size;
```

```
51      double Jfactor;//representing the strength of mutual interaction
52      double Hfactor;//representing the out field strength
53      double KbT;//beta could calculated by = pow(KbT, -1)
54
55      double RealTemperature;
56      //Just for further use, we mainly use KbT for input and calculation.
57      //Because nearly all the literature would use T*(KbT) rather than
           Real T
58
59      //Since calculate average is only two lines of code, why we choose
           to build another funtion?
60      //Why not just write it in the CalculateEnergyM's function body?
61      //The reason is: For the initial state it is ok, but later in the
           program
62      //we may gain the total values through other approaches, for example
           ,
63      //If we have calculated the changed energy, the best way is to just
           add the changed quantity to
64      // the current value, instead of calculate the whole system.
65      // in the print function we need to use CalculateAverage singly.
66      void CalculateEnergyM();
67
68      //Calculate the changed energy if a site is flipped
69      double SingleChangedEnergy(int row, int column);
70
71      //Return the spin of neighboring cells
72      //Please note, our direction is defined by:
73      //from up to down, the row number is increasing
74      //from left to right, the column number is increasing
75      int RightSpin(int row, int column);
76      int LeftSpin(int row, int column);
77      int UpSpin(int row, int column);
78      int DownSpin(int row, int column);
```

```
79      int TotalNeighborSpins(int row, int column);
80
81      //Overload all the four direction functions-Since Wolff algrithom
           needs the coordinates of points
82      //When using these four functions, the last two variables are
           designed for passing reference
83      int RightSpin(int row, int column,int &ReturnRow, int &ReturnColumn)
           ;
84      int LeftSpin(int row, int column, int &ReturnRow, int &ReturnColumn)
           ;
85      int UpSpin(int row, int column, int &ReturnRow, int &ReturnColumn);
86      int DownSpin(int row, int column, int &ReturnRow, int &ReturnColumn)
           ;
87  };
```

2 Single spin flip

Single Spin Flip is a plan for applying MCMC in ising model. The Markov process in Single Spin Flip is: we only choose one atom in one step, and decide whether we flip it by **a certain probability**. So the first requirement in 1.1.3 is easy to be satisfied: any two micro states could reach each other, although the probability may be very low.

Then what's important is the second requirement, Equation 6:

$$P_\mu \cdot P_{\mu\to\nu} = P_\nu \cdot P_{\nu\to\mu} \tag{6 revisited}$$

Since it is indeed a physical system where Importance Sampling by partition function could work, we have:

$$\frac{P_\nu}{P_\mu} = e^{-\beta(E_\nu - E_\mu)} \tag{10}$$

Then the detailed balance requirement becomes:

$$\frac{P_{\mu\to\nu}}{P_{\nu\to\mu}} = e^{-\beta(E_\nu - E_\mu)} \tag{11}$$

So the *real flip probability* needs to satisfy the above condition. There may be many plans, but Metropolis has given a most effective one.

2.1 One of the most famous MCMC: Metropolis Algorithm

Let the *real flip probability* to be named as $A(\Delta E)$, means the acceptance rate. Then this plan indeed satisfies the detailed balance condition:

$$A(\Delta E) = \begin{cases} 1 & \Delta E < 0 \\ e^{-\beta \Delta E} & \Delta E > 0 \end{cases} \tag{12}$$

It means if the energy is lower, then we must flip it, which is corresponding to our physics knowledge; even the energy is higher, there is still a little possibility to be flipped, while the higher energy of the latter state would make the possibility lower. Since β is connected to temperature, so for a certain changed energy $\Delta E(> 0)$, if the temperature is higher, the flip is more likely to happen. It also has satisfied our expectation: higher temperature would help to increase the total energy more compared to the lower temperature case.

2.2 implementation of SSF

The introduction to all the functions is shown in the Listing 1. But here we could emphasize several important points.

Destruction function for canvas class Usually we don't need to do anything in the destruction function, but this time we have used the pointer pointing to 2 dimensional array, which is allocated space by "new" function. So the class could not release those space by themselves, we need to add the "delete" function in destruction function to delete them by hand, like what I did in Listing 2.

Listing 2: **destruction function in canvas.cpp**

```
1 canvas::~canvas()
2 {
3     for (int i = 0; i < size; i++)
4     {
5         delete[]grid[i];
6     }
7     delete[]grid;
8 }
```

2.2.1 Core part in SSF

The code for Single Spin Flip is quite simple as I wrote in Listing 3, whose major principle has already been described in 2.1.

Listing 3: **single spin flip function in canvas.cpp**

```
1  //The return value does not represent whether the deltaE is positive,
       but whether the flip is successful
2  bool canvas::SingleSpinFlip(int row, int column)
3  {
4      SingleStep += 1;
5      double DeltaE = SingleChangedEnergy(row, column);
6      if (DeltaE<0)
7      {
8          //cout << DeltaE << endl;
9          grid[row][column] *= -1;
10         TotalEnergy += DeltaE;
11         TotalM += grid[row][column] * 2;
12         //cout << "yes" << endl;
13         return true;
14     }
15     else
16     {
17
18         if ((rand() % 10000)<exp(-DeltaE/KbT)*10000)
19         {
20             //cout << DeltaE << endl;
21             grid[row][column] *= -1;
22             TotalEnergy += DeltaE;
23             TotalM += grid[row][column] * 2;
24             //cout << "yes" << endl;
25             return true;
26         }
27     }
```

```
28      return false;
29  }
```

2.3 Result of SSF

2.3.1 The condition of Markov Chain requires convergence

As 1 shows, there are three generation ways: all positive, all negative or random. From 1.2.1 we could know for the model we considered, all positive and all negative is equivalent:

$$H = -\sum_{i \text{ and } j \text{ are the 1st nearest neighbor}} \sigma_i \sigma_j \qquad \text{(9 revisited)}$$

Both of them means the system is starting from the lowest energy state, while the random state is corresponding to the highest energy state. So one method to decide whether the algorithm has converged is to compare the energy of random and positive/negative: from the zeroth law of thermodynamics [4], if they have the same energy under the same temperature, then both of them have converged.

But why we need to converge? That's because from the definition of MCMC, we know the sample following the statistical property only when it has reached steady state. So not converging would result in the failure of sampling, thus the failure of MCMC result.

Since all positive is equivalent to all negative, so I just choose all positive to represent the lowest energy state. Figure 1 is the result of all positive and random after 100000 monte carlo steps(1 billion single steps for 100 by 100 grid). We can see their results are very close, so it has met our expectation.

Here comes one concept: **Monte Carlo Step** is equal to the total single spin flip steps divided by the whole space, here 100 by 100 = 10000.

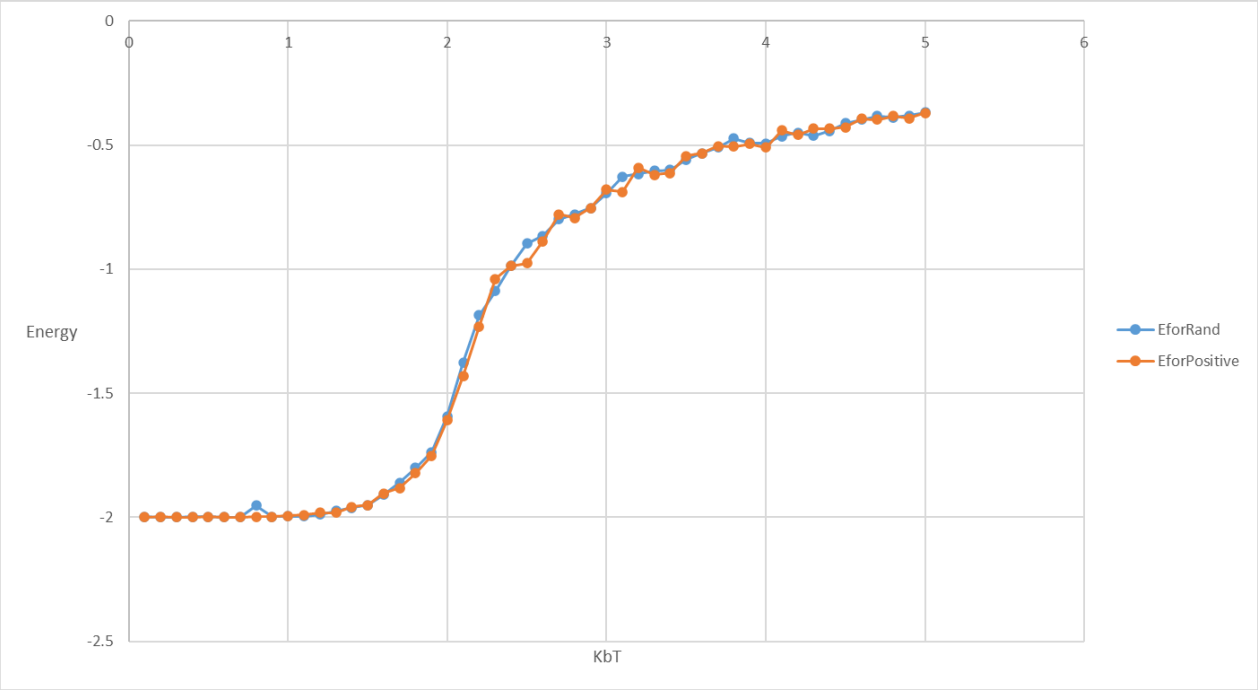


Figure 1: Energy for Positive and Rand after 1billion steps for 100by100 namely 100000 mcs

2.3.2 Getting the critical temperature

Then we could show the result of AverageM of all positive and random to show the phase transition(Figure 2), here we choose the temperature range to be 0.1 to 5, and we could find the phase transition happened in the range 2 to 2.5, so we could focus on this range to carry out more precise experiment.(Prof. Zhu has told us this idea works for all computational problems, many years after we may forget the details of the knowledge, but we should remember this idea: carrying out a rough experiment firstly then focus on the important range to get explicit value)

By the way, we can see there is a bad point whose KbT is 0.8 in Figure 2, we would print its configuration of the final state to explain it in subsubsection 2.3.3.

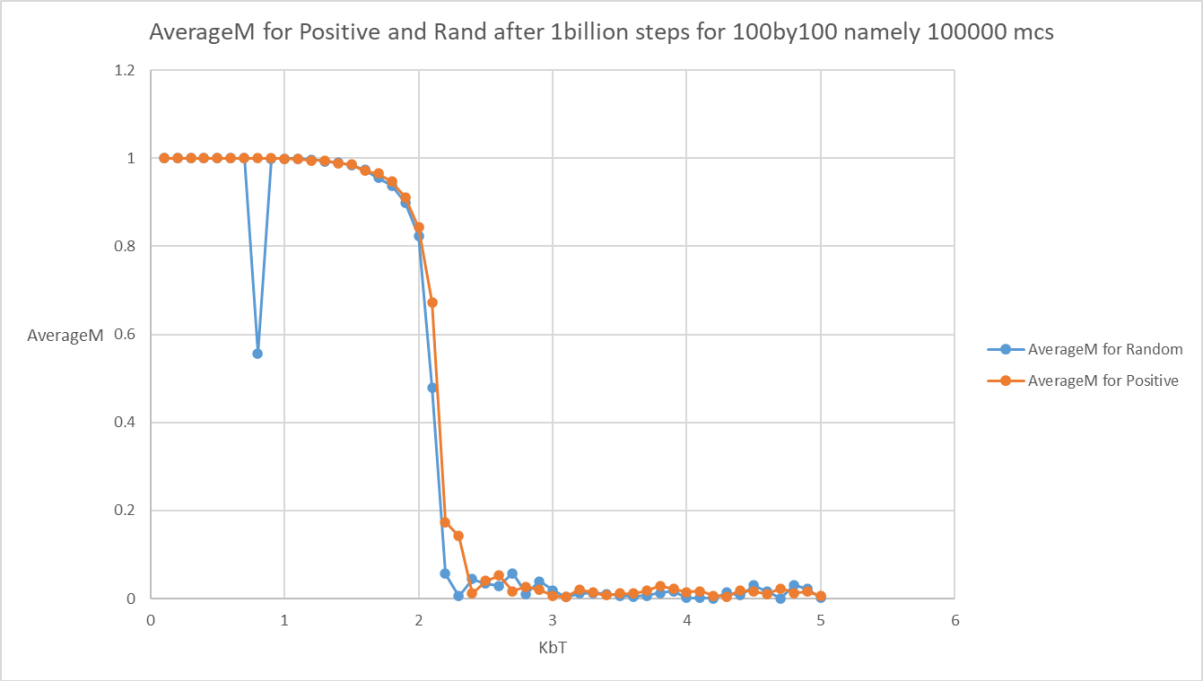


Figure 2: AverageM for Positive and Rand after 1billion steps for 100by100 namely 100k mcs

For the important range 2 to 2.5, I have carried out the experiment still 1 billion single steps(100 kilo monte carlo steps) for every 0.01 KbT. Both of all positive and random initialization have been tested.

From Figure 3 we could read out that the sudden change of Average Magnetic susceptibility happened between points 2.10 and 2.12 in the all positive initialization case(2.11 is even less than 2.12, but considering the value of the latter points, 2.12 is better to be the end of the sudden change), so the experimental critical temperature is assumed to be 2.11.

From Figure 4 we could read out that the sudden change of Average Magnetic susceptibility happened between point 2.09 and 2.10 in the random initialization case, so the experimental critical temperature is assumed to be 2.095.

Take an average of the two experimental values, the critical temperature we got from SSF method is KbT=(2.11+2.095)/2=2.1025.

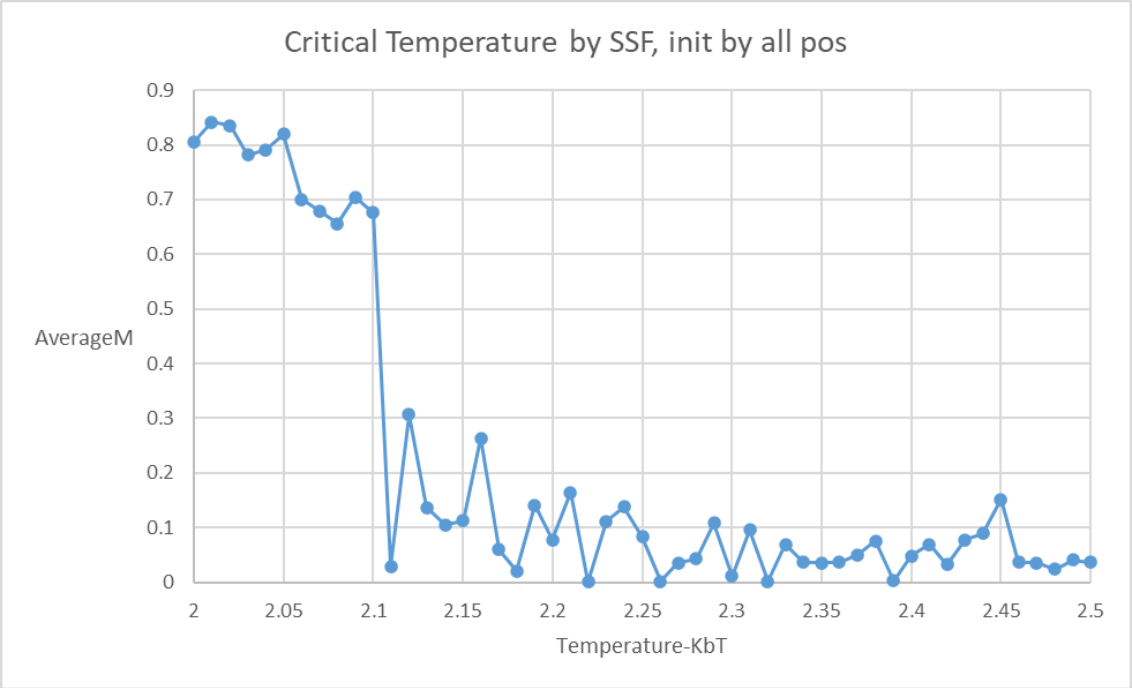


Figure 3: critical temperature by SSF, init by all pos

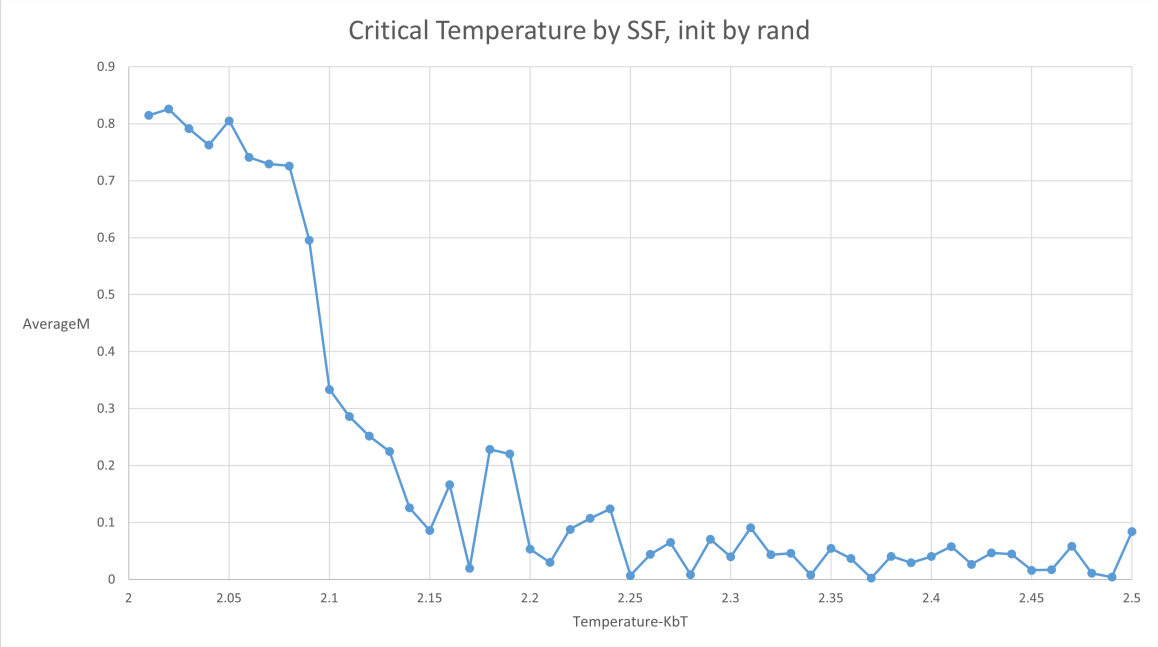


Figure 4: critical temperature by SSF, init by rand

2.3.3 The weakness of SSF from the result:may be stuck in metastable state

As we have said in the previous part, there is a bad point in Figure 2. Its left and right neighbors are all converged while it has not converged.

Its configuration could be seen in 5, the reason could be concluded as **SSF may be stuck in Metastable state**.

Metastable state means the whole space is divided into two parts with straight boundaries. In this case, the changed energy of all the points is larger than zero thus hard to be flipped:

If the SSF chooses the point far away from the boundary, then it would be very difficult to be flipped since the changed energy would be +8; even the point on the boundary has a changed energy=+4.

We know that there is some little possibility to be flipped even the changed energy is negative, so let us suppose we are very lucky to flip all the (size) points along the boundary which means we need to succeed to flip the positive changed energy points continuously for (size) times. But even so, if the next flip was on the opposite direction compared to our previous (size) flips, then the configuration would turn back to our previous one(since the atoms on the boundary are easier to be flipped than the non-boundary ones, though still very difficult).

Wolff could solve this problem, because it is cluster flip instead of single flip, which we would discuss in the next section.

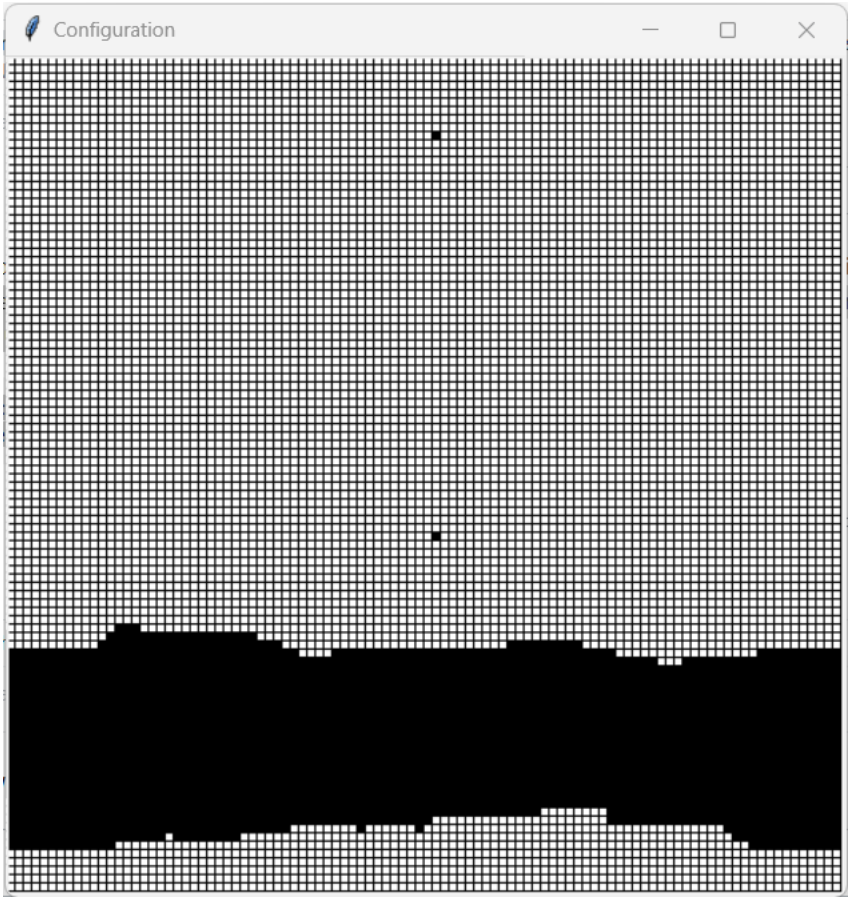


Figure 5: Example of Metastable state: Configuration of KbT=0.8 1Billion times SSF randomly initialization

3 Wolff Algorithm

Wolff algorithm is a kind of cluster flip algorithm [5] which could flip a cluster of atoms at once, thus could have better efficiency than the single spin flip algorithm.

Still, Wolff algorithm is a kind of Markov Chain Monte Carlo Algorithm, so it needs to satisfy the two conditions of MCMC. And it is still in the system where Importance Sampling by partition function could work, so (11) still holds.

$$\frac{P_{\mu\to\nu}}{P_{\nu\to\mu}} = e^{-\beta(E_\nu - E_\mu)} \qquad \text{(11 revisited)}$$

And the process of Wolff Algorithm becomes:

1. One atom was selected randomly in the whole system. Give it a name "seed atom".
2. Start to poll over all the neighbor atoms with the same spin with the "seed atom": decide whether they should be added to the cluster by the probability P_{adding}
3. For everyone in the cluster, but has not been chosen as the "seed atom", select it to be the "seed atom" and operate the 2nd step to add its neighbors. This procedure would terminate when there is no atom which has not been selected to be the "seed atom".
4. Decide whether the whole cluster should be flipped by a probability $A(\Delta E)$, called the acceptance rate.

And just what we have done in the single spin flip, now we should express the LHS of 11 by the quantities in Wolff Algorithm.

So suppose there are two states ν and μ, we could have:

$$\frac{P_{\mu\to\nu}}{P_{\nu\to\mu}} = \frac{g_{\mu\to\nu}A_{\mu\to\nu}}{g_{\nu\to\mu}A_{\nu\to\mu}} = e^{-\beta(E_\nu - E_\mu)} \tag{13}$$

It shows that there are many plans, that is, many $g \leftrightarrow A$ pairs could satisfy our requirement.

Can we make some further simplification? Yes! Because the first part of P, namely the g part, could be described as two steps:

1. Firstly, we need to let all the atoms in the cluster could be added into the cluster successfully. So if the probability of the first atom is chosen to be the "seed atom" is P_{init}, and there are $x-1$ other atoms are added, then we need to be as lucky as ${P_{init} \cdot P_{adding}}^{x-1}$
2. Secondly, all the neighbor atoms of the cluster members, have to be rejected to the cluster. If the number is m, then we need to be as lucky as $(1 - P_{adding})^y$

One surprising thing is, due to the process of the Wolff method, the first step is just the same for μ and ν, so when we consider the ratio $\frac{g_{\mu\to\nu}}{g_{\nu\to\mu}}$, the first step would cancel! So only the second step would remain. Suppose the y to be y_μ and y_ν for the correct and inverse processes, then we have

$$\frac{g_{\mu\to\nu}}{g_{\nu\to\mu}} = (1 - P_{adding})^{y_\mu - y_\nu} \tag{14}$$

Back to Equation 13 we have:

$$\frac{A_{\mu\to\nu}}{A_{\nu\to\mu}} = e^{-\beta(E_\nu - E_\mu)} \cdot \frac{g_{\nu\to\mu}}{g_{\mu\to\nu}} = e^{-\beta(E_\nu - E_\mu)} \cdot (1 - P_{adding})^{y_\nu - y_\mu} \tag{15}$$

A more surprising thing is, Wolff model is so delicate that there is a relationship between $y_\nu - y_\mu$ and $E_\nu - E_\mu$:

$$E_\nu - E_\mu = 2J(y_\mu - y_\nu) \tag{16}$$

Then we could have:

$$\frac{A_{\mu\to\nu}}{A_{\nu\to\mu}} = \left[e^{2\beta J}(1 - P_{adding})\right]^{y_\nu - y_\mu} \tag{17}$$

Then we could find, there is a very special solution could let the acceptance rate to be independent of $y_\nu - y_\mu$: just let P_{adding} to be $1 - e^{-2\beta J}$, then the base number could be reduced to 1!

$$\frac{A_{\mu\to\nu}}{A_{\nu\to\mu}} = 1 \tag{18}$$

So if $P_{adding} = 1 - e^{-2\beta J}$, the acceptance rate could be a constant. For simplicity, we could just select it to be 1 since $\frac{1}{1} = 1$, to let the algorithm perform better(this behavior is just corresponding to what Metropolis does in single spin flip, there are also many acceptance rate for SSF and he chose the best one):

$$P_{adding} = 1 - e^{-2\beta J}, \quad A(\Delta E) = 1 \tag{19}$$

3.1 implementation of Wolff

The core function for Wolff is in Listing 4. Then let me stress some important designs separately, some of which let me understand the Wolff Algorithm better.

Listing 4: **Wolff function in canvas.cpp**

```
1 int canvas::WolffFlip(int row, int column)
2 {
3     //We put the init of flag here because there is not a rule that one
          could only be flipped once
4     bool ** flag = new bool* [size];
```

```
5       for (int i = 0; i < size; i++)
6       {
7           flag[i] = new bool[size];
8           for (int j = 0; j < size; j++)
9           {
10              flag[i][j] = false;//When it is false, it has not been
                    flipped by Wolff
11          }
12      }
13      flag[row][column] = true;
14      int ClusterSize = 1;
15
16      double P_adding = 1-exp(-2 * Jfactor / KbT);
17
18      queue<vector<int>> Queue1;
19      vector<int>* QueueElement;
20      QueueElement = new vector<int>{ row, column };
21      Queue1.push(*QueueElement);
22      delete QueueElement;
23
24      while (!Queue1.empty())
25      {
26          //I do not want to write another function, so just new two
                variables as "input"
27          int i;
28          int j;
29          vector<int> Temp = Queue1.front();
30          Queue1.pop();//When poping an element, it would be deleted
                automatically, so "delete" operation is not compulsory here
31          i = Temp[0];
32          j = Temp[1];
33
34          int NeighborRow, NeighborColumn;
```

```
35          //Here the spin function must be written before the flag list
               judgement, since the Neighbor coordinates are assigned values
                in the spin functions
36          if (RightSpin(i,j,NeighborRow,NeighborColumn)==grid[i][j]&&flag[
               NeighborRow][NeighborColumn]==false)
37          {
38              if (rand()%10000<P_adding*10000)
39              {
40                  QueueElement = new vector<int>{ NeighborRow,
                       NeighborColumn };
41                  Queue1.push(*QueueElement);
42                  flag[NeighborRow][NeighborColumn] = true;
43                  ClusterSize += 1;
44                  delete QueueElement;
45              }
46          }
47          if (LeftSpin(i, j, NeighborRow, NeighborColumn) == grid[i][j] &&
                flag[NeighborRow][NeighborColumn] == false)
48          {
49              if (rand() % 10000 < P_adding * 10000)
50              {
51                  QueueElement = new vector<int>{ NeighborRow,
                       NeighborColumn };
52                  Queue1.push(*QueueElement);
53                  flag[NeighborRow][NeighborColumn] = true;
54                  ClusterSize += 1;
55                  delete QueueElement;
56              }
57          }
58          if (UpSpin(i, j, NeighborRow, NeighborColumn) == grid[i][j] &&
               flag[NeighborRow][NeighborColumn] == false)
59          {
60              if (rand() % 10000 < P_adding * 10000)
```

```
61              {
62                  QueueElement = new vector<int>{ NeighborRow,
                        NeighborColumn };
63                  Queue1.push(*QueueElement);
64                  flag[NeighborRow][NeighborColumn] = true;
65                  ClusterSize += 1;
66                  delete QueueElement;
67              }
68          }
69          if (DownSpin(i, j, NeighborRow, NeighborColumn) == grid[i][j] &&
                flag[NeighborRow][NeighborColumn] == false)
70          {
71              if (rand() % 10000 < P_adding * 10000)
72              {
73                  QueueElement = new vector<int>{ NeighborRow,
                        NeighborColumn };
74                  Queue1.push(*QueueElement);
75                  flag[NeighborRow][NeighborColumn] = true;
76                  ClusterSize += 1;
77                  delete QueueElement;
78              }
79          }
80
81          //Do the flip for the element just poped
82          //Why now? Because we need to use its grid value before
83          //Flip every one in its loop is equivalent to flip all the
                cluster at once, and former method consumes less computation
                resource
84          TotalEnergy += SingleChangedEnergy(i, j);
85          grid[i][j] *= -1;
86          TotalM += grid[i][j] * 2;
87      }
88
```

```
89      for (int i = 0; i < size; i++)
90      {
91          delete[]flag[i];
92      }
93      delete[]flag;
94
95      return ClusterSize;
96  }
```

3.1.1 About the 2D array recording the flip information

We use a 2D array's pointer to record whether the atom has been flipped. As Listing 5 shows, we put its initialization in Wolff function instead of for every Canvas object. It corresponds to the fact that every atom could only be flipped once during one Wolff step, but could be flipped many times during the whole program.

What's more, just like what we did in the destruction function of Canvas in canvas.h, we should delete the 2D array at the end of every Wolff step (Listing 6).

Listing 5: **flag initialization in Wolff function in canvas.cpp**

```
1  int canvas::WolffFlip(int row, int column)
2  {
3      //We put the init of flag here because there is not a rule that one
           could only be flipped once
4      bool ** flag = new bool* [size];
5      for (int i = 0; i < size; i++)
6      {
7          flag[i] = new bool[size];
8          for (int j = 0; j < size; j++)
9          {
10             flag[i][j] = false;//When it is false, it has not been
                   flipped by Wolff
11         }
12     }
```

Listing 6: **flag deleting in Wolff function in canvas.cpp**

```
    for (int i = 0; i < size; i++)
    {
        delete[]flag[i];
    }
    delete[]flag;

    return ClusterSize;
}
```

3.1.2 Overloading the four functions: RightSpin, LeftSpin, UpSpin, DownSpin

These four functions have also appeared in SSF, where we used them to get the spin value of the neighbor atom. But in Wolff Algorithm, we need to get the coordinates of the neighbor atoms. To avoiding causing too many extra functions, I just used overload, the property of C++, to make the four functions applicable for Wolff Algorithm.

Besides, there are two things we need to notice:

1. unlike the spin value, the coordinates have two values x and y to be returned
2. overloading property requires the function with the same name should have different input variables to be differentiated from the old ones.

For the first problem, we could change the return value of the functions from int to the pointer of int, but overloading in C++ does not permit the difference between initial function and the overloading function is only return value. So this idea is not good.

And for the second problem, of course we could make some "useless" variables to solve.

But we could have a better choice to solve the two problems at the same time: setting two extra input variables as the parameters of the function, which are passing their reference instead of passing the value. The example of RightSpin could be seen in Listing 7 while the overloading one could be seen in Listing 8

Listing 7: **Initial RightSpin function in canvas.cpp**

```
int canvas::RightSpin(int row, int column)
```

```
{
    int RightColumnNum;
    if (column<size-1)
    {
        RightColumnNum = column + 1;
    }
    else if(column==size-1)
    {
        RightColumnNum = 0;
    }
    return grid[row][RightColumnNum];
}
```

Listing 8: **Overloading RightSpin function in canvas.cpp**

```
int canvas::RightSpin(int row, int column, int & ReturnRow, int &
   ReturnColumn)
{
    int RightColumnNum;
    if (column < size - 1)
    {
        RightColumnNum = column + 1;
    }
    else if (column == size - 1)
    {
        RightColumnNum = 0;
    }
    ReturnRow = row;
    ReturnColumn = RightColumnNum;
    return grid[row][RightColumnNum];
}
```

3.1.3 An equivalent operation: flip every atom after its neighbors have all been enquired ⇔ flip the whole cluster finally

We select this position to flip the atom in the cluster. Firstly we could know it is equivalent to flipping all the cluster at the end of program: because in Wolff Algorithm we have chosen the plan whose acceptance rate is 1(19). Secondly, if we update the whole cluster at the end, then we need to poll over all the atoms to check whether it has the sign of "in the cluster". Since cluster's size is always less than the whole space, this choice would definitely waste more time.

Listing 9: **Judge right neighbor in Wolff in canvas.cpp**

```
1         //Do the flip for the element just poped
2         //Why now? Because we need to use its grid value before
3         //Flip every one in its loop is equivalent to flip all the
              cluster at once, and former method consumes less computation
              resource
4         TotalEnergy += SingleChangedEnergy(i, j);
5         grid[i][j] *= -1;
6         TotalM += grid[i][j] * 2;
```

3.1.4 FIFO working better for cluster growing mechanism: we could give a rough proof

At any time, the atoms existing in the whole space could be divided into:

1. The atoms which have not been added into the cluster.
2. The atoms which have been added into the cluster.
   (a) The atoms whose all the neighbors have already been asked to add or not
   (b) The atoms which has not been asked about neighbors

So the 2D array flag pointing to could only help us differentiate the 1 and 2. But for 2(a) and 2(b), we need something more to differentiate.

C++11 has offered plenty of containers in its STL(standard template library) [6], often used

ones among which are *vector*, *queue* and *stack*. Here we want to use a container to contain 2(b), which would be removed from the container when they become 2(a). And since the growing trend of the cluster is from a single atom spreading to surrounding atoms, it is very natural to think of the idea to use *queue* which is **FIFO(first in first out)** rather than *stack* which is **LIFO(Last-In, First-Out)**. But does FIFO really work better than LIFO here? We could give a rough **proof**:

In the function body of Wolff algorithm, when we are enquiring the four neighbors in turn, we would repeat this judgement four times. Take the right neighbor as an example(**Judge right neighbor in Wolff in canvas.cpp**), there are two conditions after **if**: the first is to check whether the neighbor shares the same spin as the seed, the second is whether it is already in the cluster. This order could not be changed because we need the overloading neighbor spin functions to get the coordinates of the neighbor, which are required in the second step's calculation.

Since the order could not be changed, we need to consider another rule in C++: if there are two or more conditions in the bracket after **if**, and they are connected by && which means "and", then C++ would test it from left to right: if any of the conditions is false, then the total result must be false and the rest conditions would not be considered anymore.

This is the point of view we could find some ways to save time. Making the condition terminate as early as possible could save us time.

For an atom in a cluster, its neighbors may be in the cluster or out of the cluster, let us say the size of the cluster is x, then the total number of the cluster's neighbors is:

$$N_{total} = N_{in} + N_{out} \tag{20}$$

If the cluster has been specified, then the outside neighbors has also been specified-no matter how we choose the sequence of the atoms in the cluster to be enquired, the outside neighbors would not change their spins, thus the cost time would not change too.

But the inner neighbors are different, if we can choose the atom which has the more inner neighbors to be the seed early, then it would be flipped early. As a result, when it is judged by others, it would give a quick result since it has been flipped compared to the other un-flipped

atoms in the cluster. So if we choose the atom with the more neighbors to be the seed atom earlier, then we would save more time!

And if we use FIFO, we could consider the inner atoms of the cluster to be seed early! By contrast, the LIFO method would cost more time! Q.E.D.

Listing 10: **Judge right neighbor in Wolff in canvas.cpp**

```
1        if (RightSpin(i,j,NeighborRow,NeighborColumn)==grid[i][j]&&flag[
             NeighborRow][NeighborColumn]==false)
```

3.2 Result of Wolff

3.2.1 Prove the simulation has converged

From what we knew in SSF part, we could select the temperature range 2 - 3 to be studied. Just like what we did in 2.3.1, if the final state's energy of All positive initialization and random initialization is close, then we could say in this temperature range, the algorithm has converged.

From Figure 6 we found 200 Wolff steps is not enough for the temperature higher than 1.9; from Figure 7 we could find 1000 Wolff steps would be valid for a higher temperature range-until 2.5, but still could not ensure convergence for 2.5 to 3; from Figure 8, **we finally made all the temperature points among 1 to 3 converge, so we could use 2000 as the simulation steps to find the critical temperature in the later part.**

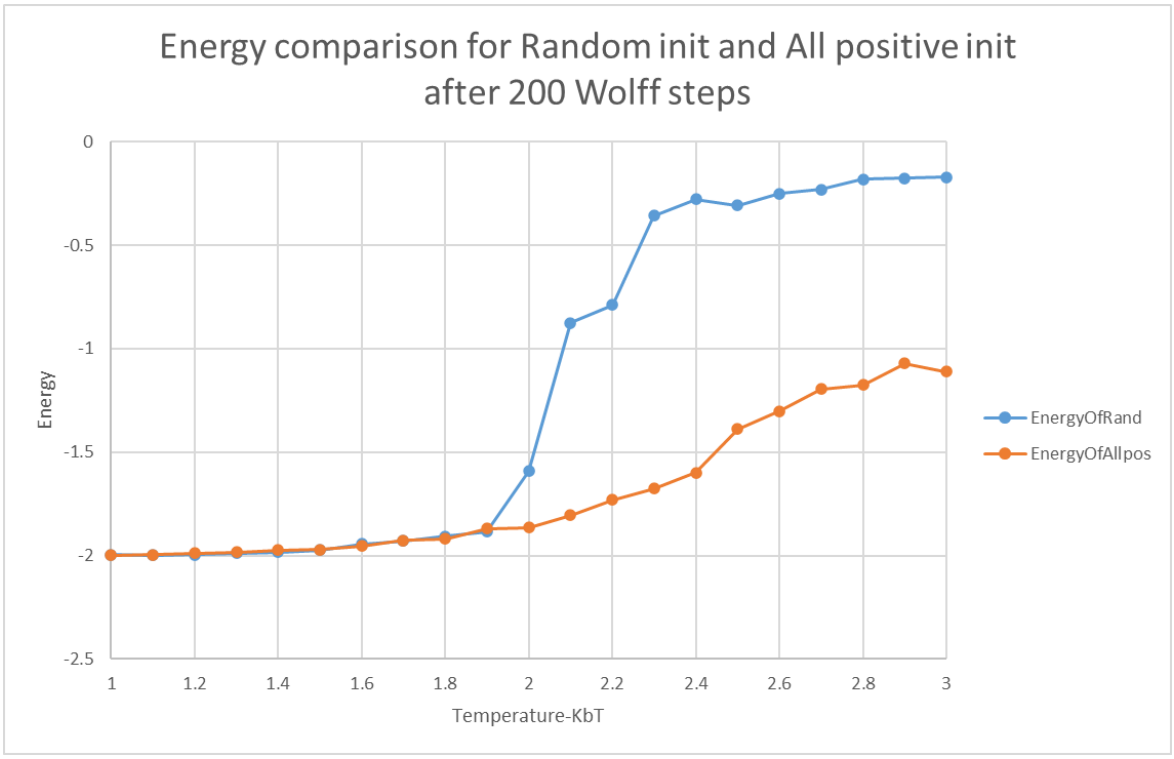


Figure 6: Energy for Positive and Rand after 200 Wolff Steps

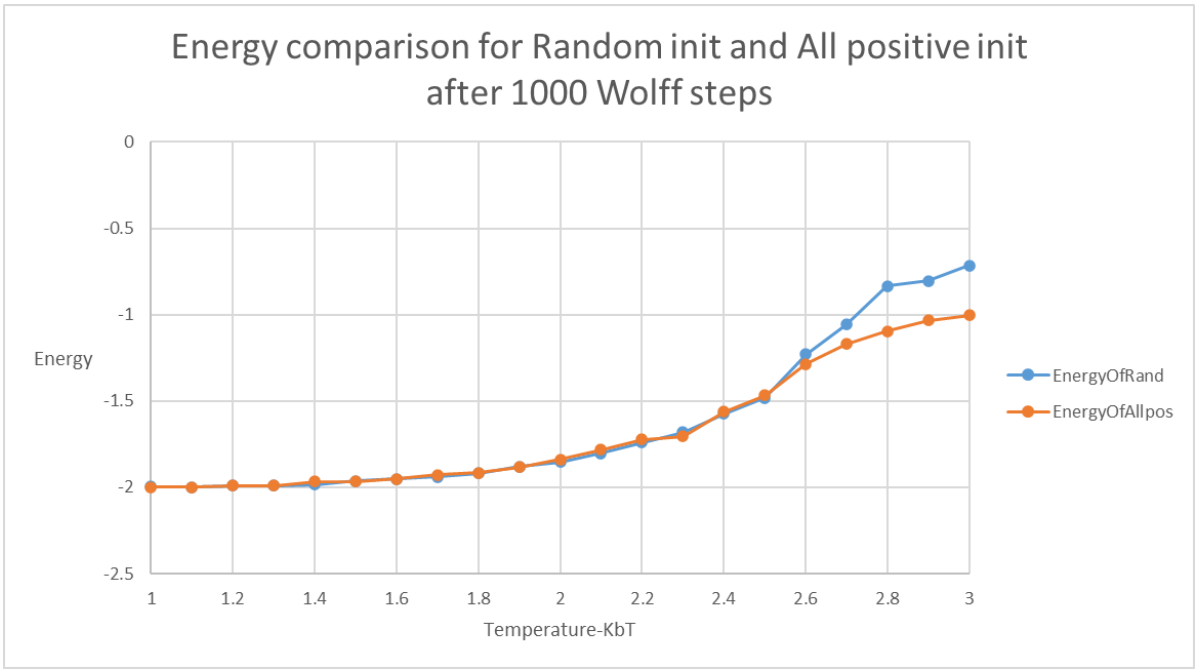


Figure 7: Energy for Positive and Rand after 1000 Wolff Steps

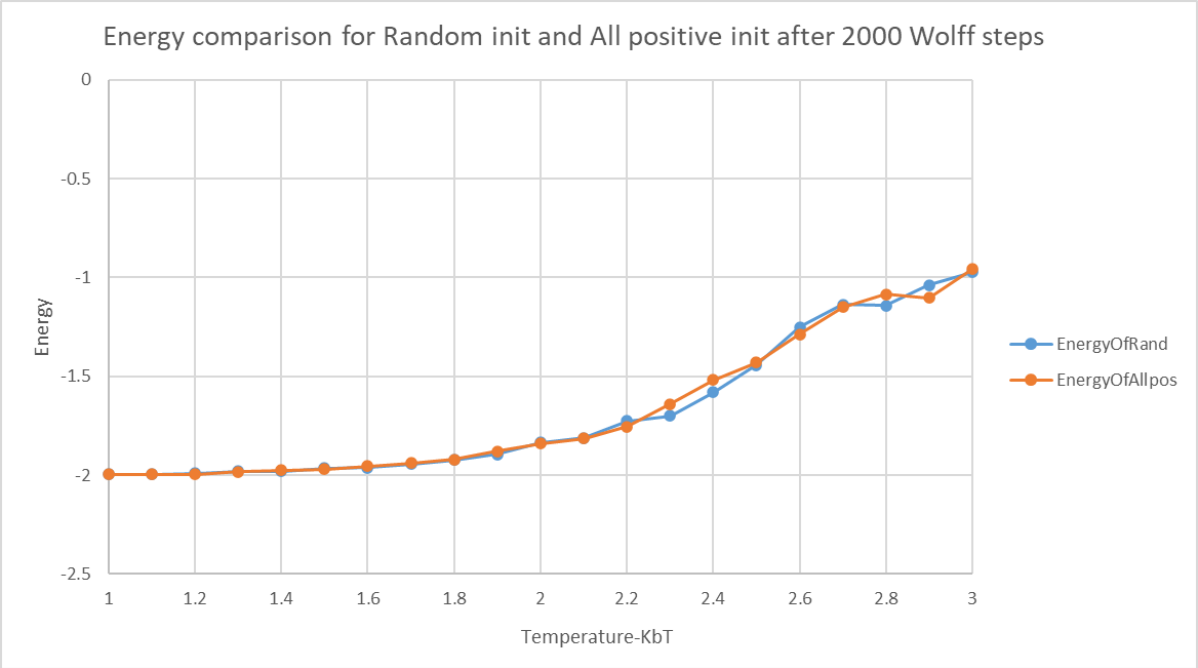


Figure 8: Energy for Positive and Rand after 2000 Wolff Steps

Then let us see the trend of Average Magnetic susceptibility for the case of 2000 Wolff steps to get a more explicit temperature range: Figure 9. From this figure we could see the sudden change lies in 2.5 to 2.6. **So we chose 2.4 to 2.7 as the explicit studied temperature range in the later part**.

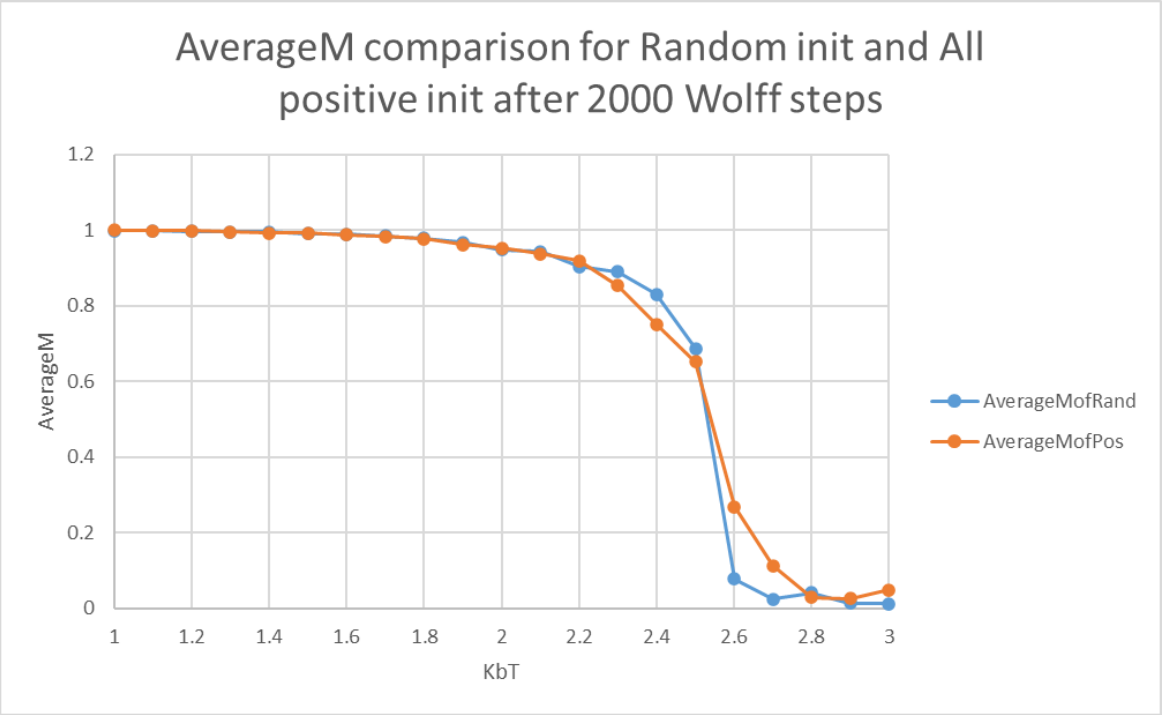


Figure 9: AverageM for Positive and Rand after 2000 Wolff Steps

3.2.2 Finding the critical temperature by Wolff

Combining the two conclusions we got in 3.2.1, we could use 2000 Wolff steps as the parameter to simulate in a more explicit temperature range 2.4-2.7. The result could be seen in Figure 10.

For the positive initialization, we could read out that the sudden change of Average Magnetic susceptibility happened between points 2.54 and 2.55 in the all positive initialization case, so the experimental critical temperature is assumed to be 2.545.

For the random initialization, we could read out that the sudden change of Average Magnetic susceptibility happened between point 2.51 and 2.52 in the random initialization case, so the experimental critical temperature is assumed to be 2.515.

Take an average of the two experimental values, the critical temperature we got from Wolff method is KbT=(2.515+2.545)/2=2.53.

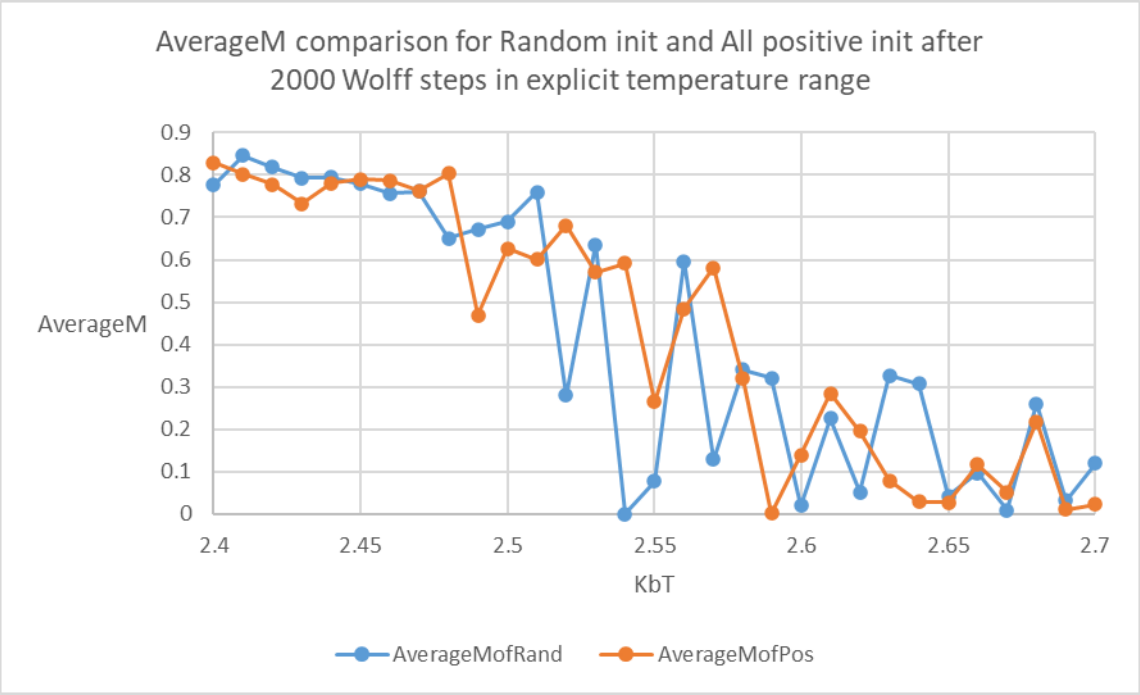


Figure 10: AverageM for Positive and Rand after 2000 Wolff Steps in explicit temperature range

3.2.3 Showing the process of Wolff by an example

Printing the configuration of every step for Wolff Algorithm may help to understand the algorithm. So I have selected two typical special cases to show the configurations:

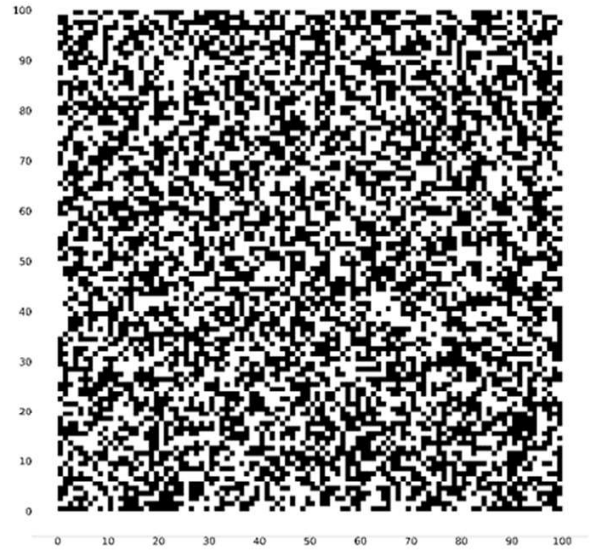
1. the environment temperature is below the critical temperature(KbT=1.5) but the system's initial temperature is high(initialized by random configuration)
2. the environment temperature is beyond the critical temperature(KbT=3) but the system's initial temperature is low(initialized by all same directions(all positive/all negative) configuration)

KbT=1.5, initialized by random We could look at the initial several states and the last several states.

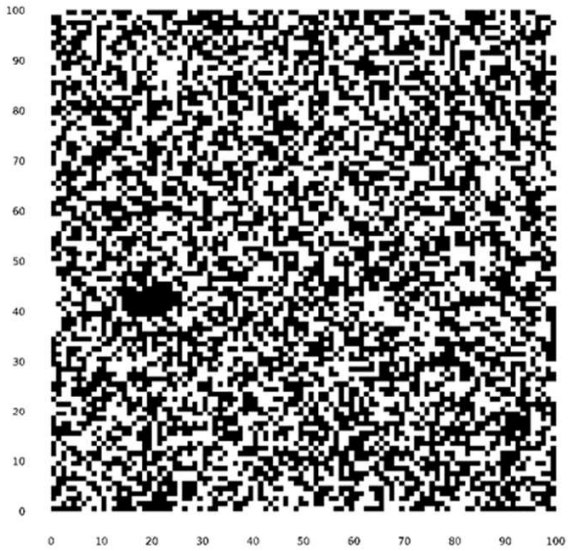
1. From the step 0-11(the next two pages pdf), we could find some small clusters have been formed in the random configuration.
2. From the step 1990-1995(the 3rd page pdf after the current page), we could find that the

Average Magnetic susceptibility is nearly 1 and the configuration appears to be nearly all positive and nearly all negative alternatively.

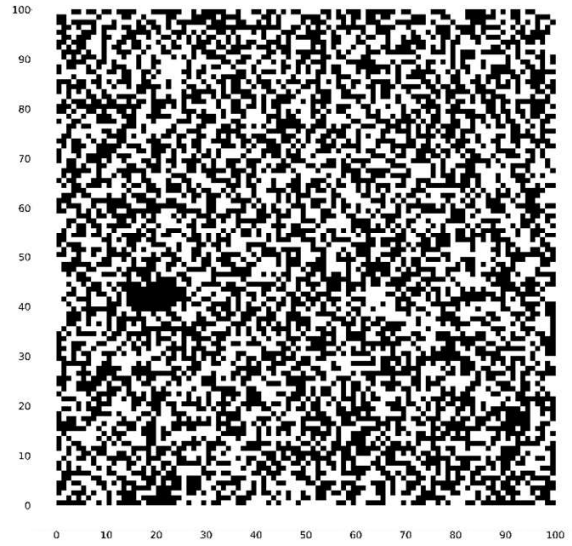
KbT=1.500000 0



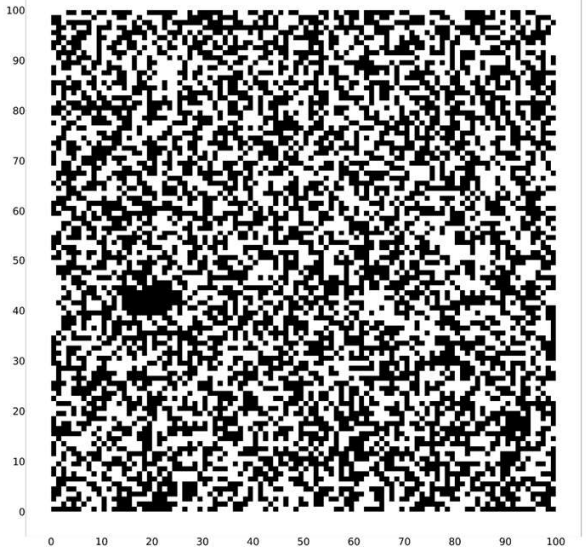
KbT=1.500000 1



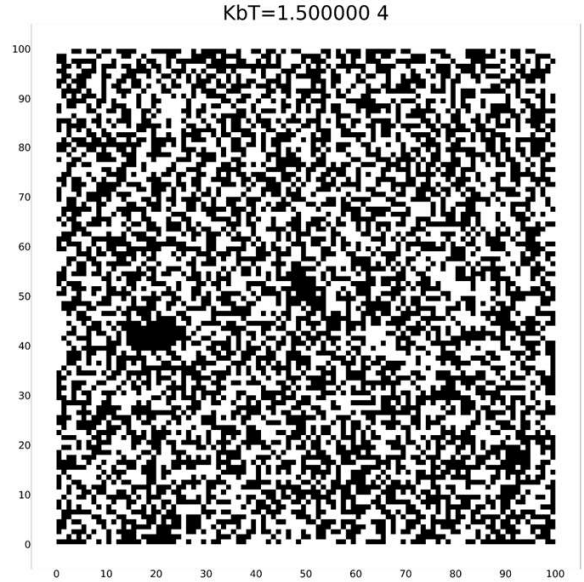
KbT=1.500000 2



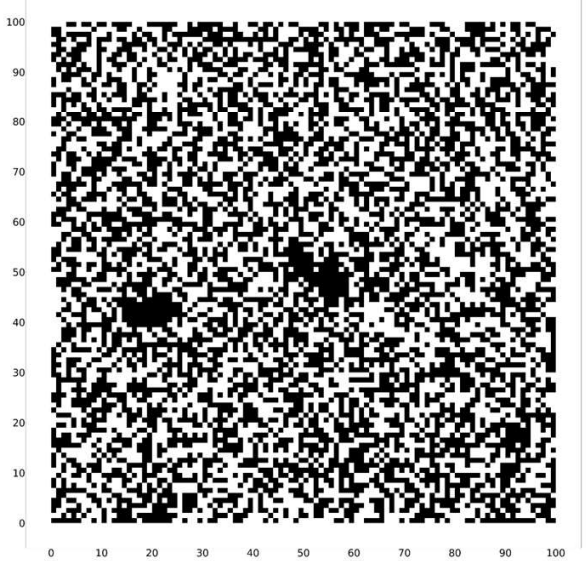
KbT=1.500000 3



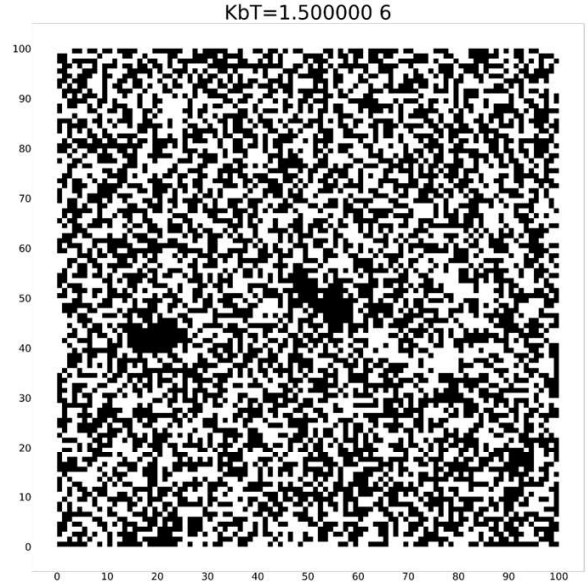
KbT=1.500000 4



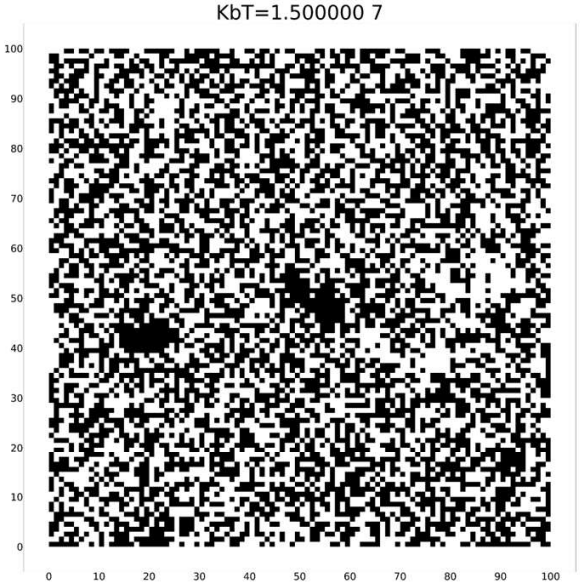
KbT=1.500000 5



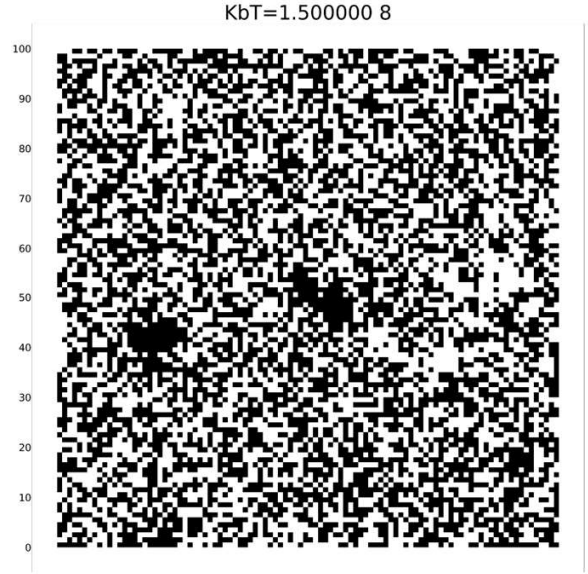
KbT=1.500000 6



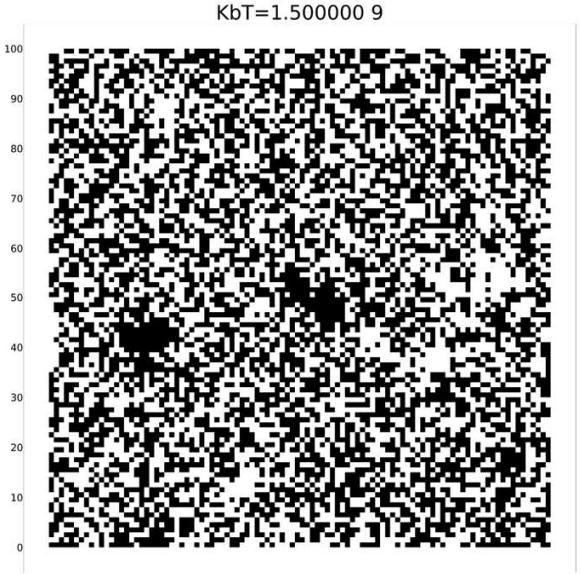
KbT=1.500000 7



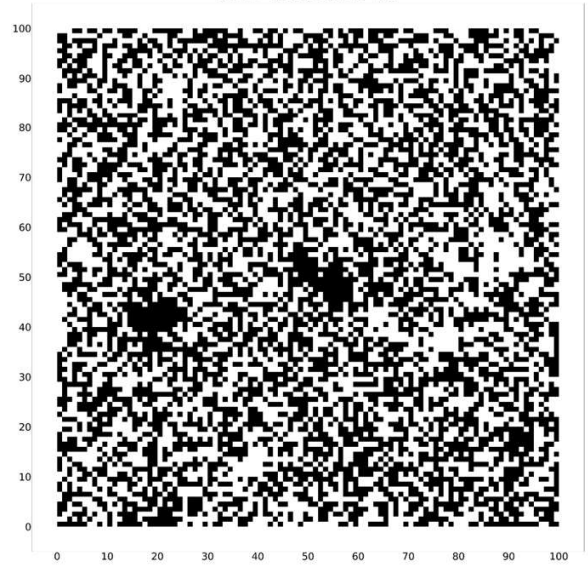
KbT=1.500000 8



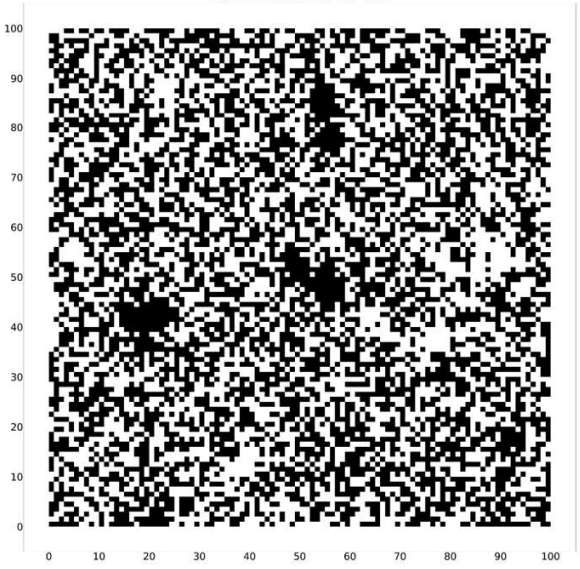
KbT=1.500000 9

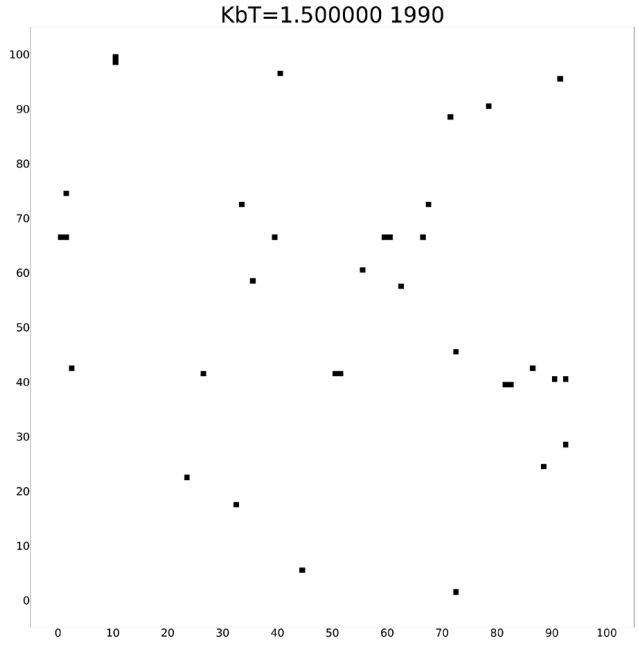


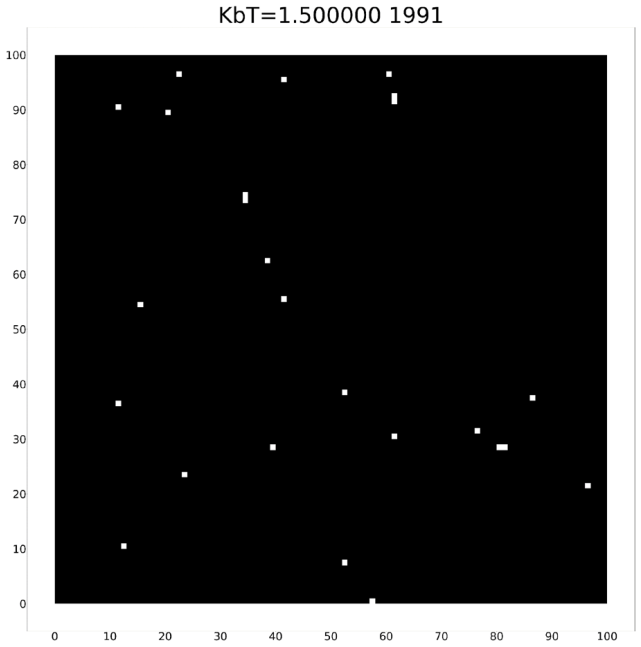
KbT=1.500000 10

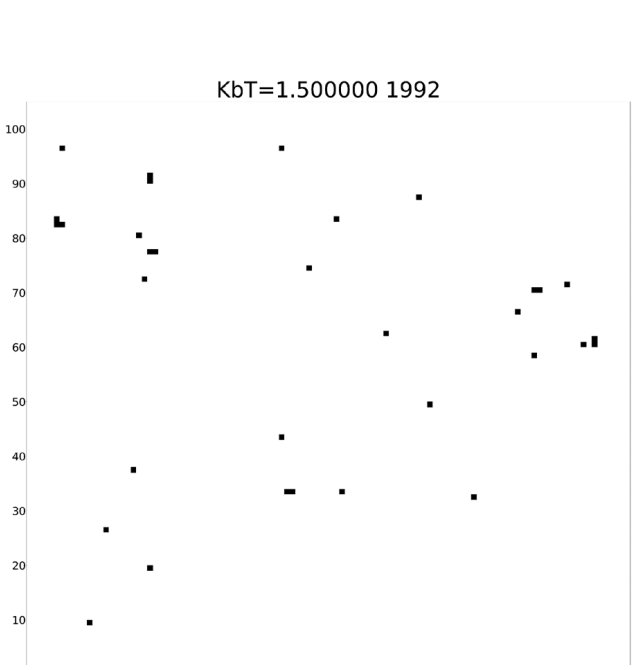


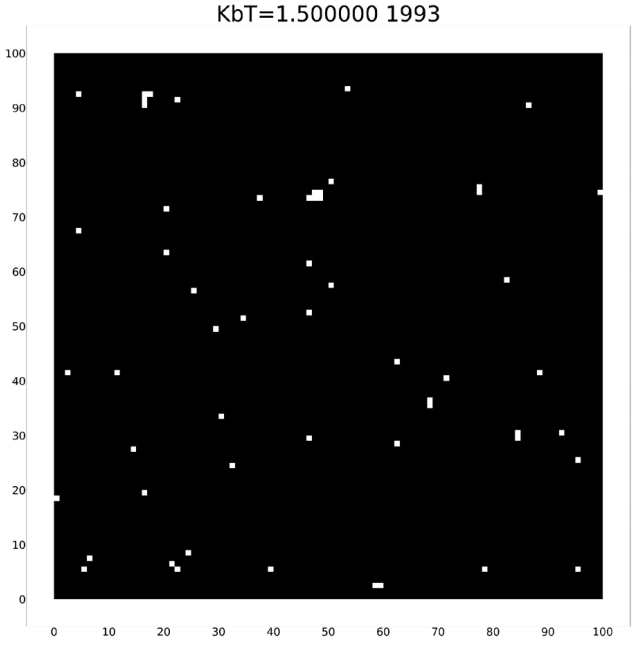
KbT=1.500000 11

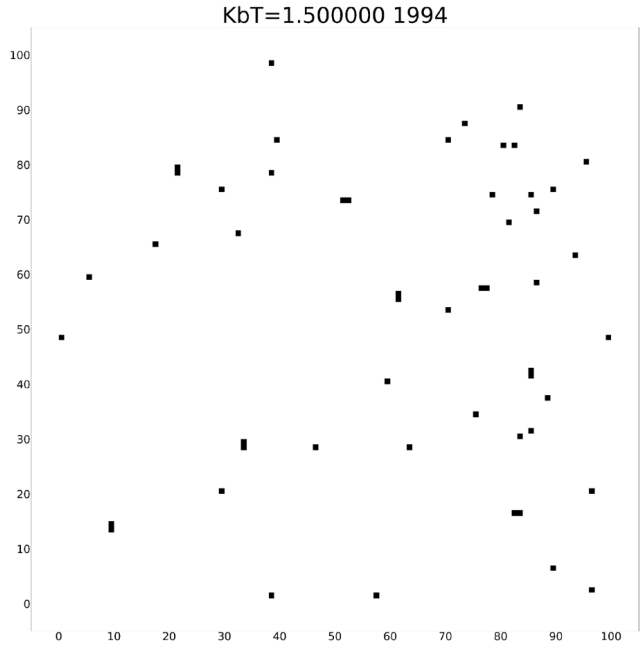


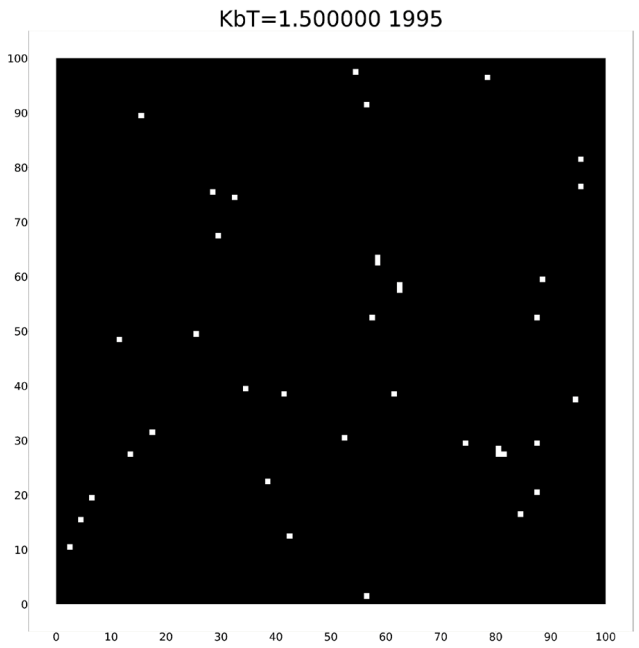

KbT=1.500000 1990


KbT=1.500000 1991


KbT=1.500000 1992


KbT=1.500000 1993

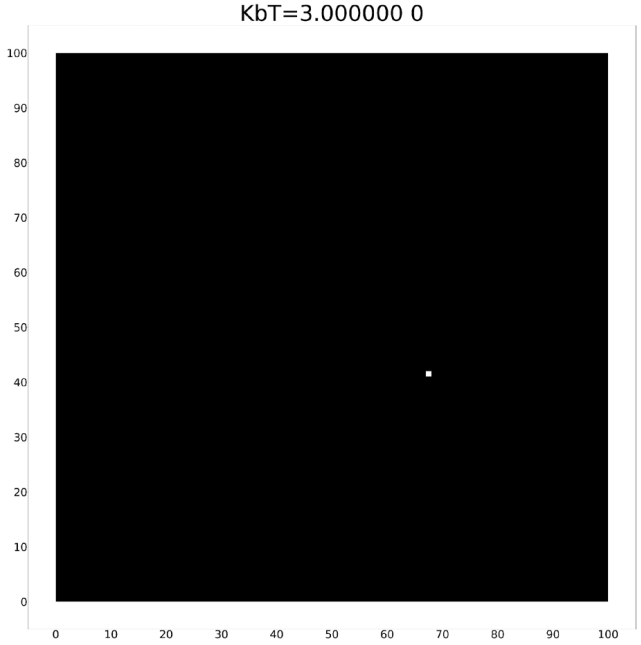

KbT=1.500000 1994


KbT=1.500000 1995

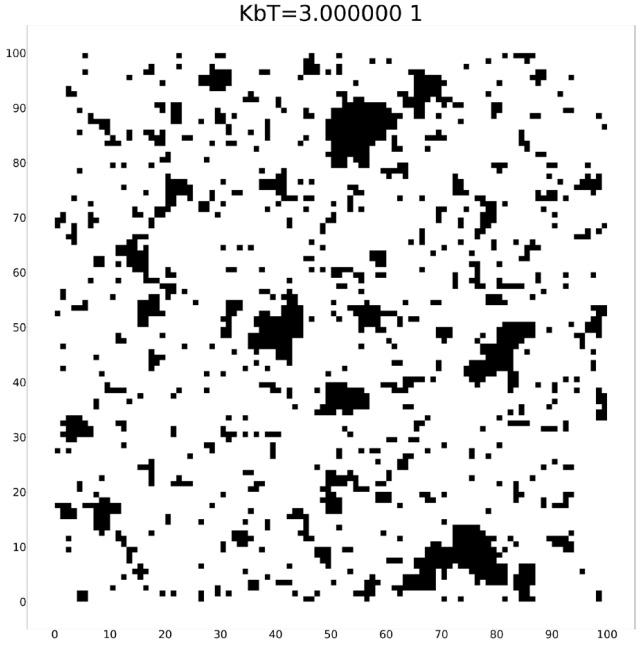
KbT=3, initialized by all positive We could look at the initial several states and the last several states too.

1. From the step 0-11(the next two pages pdf), we could find some clusters have been formed quickly.
2. From the step 1990-1995(the 3rd page pdf after the current page), we could find that the Average Magnetic susceptibility is still nearly 0(randomly configuration).

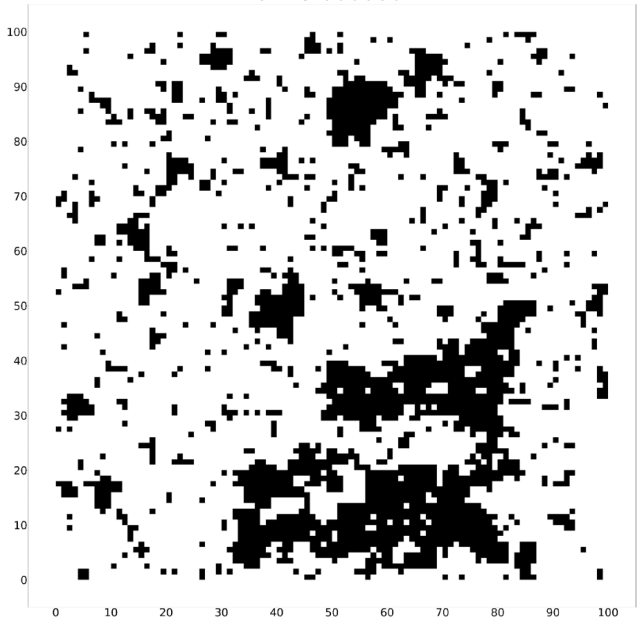
KbT=3.000000 0



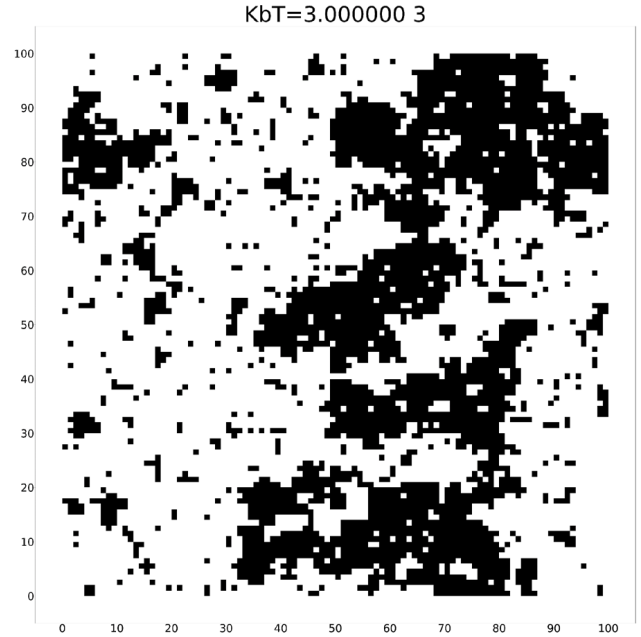
KbT=3.000000 1



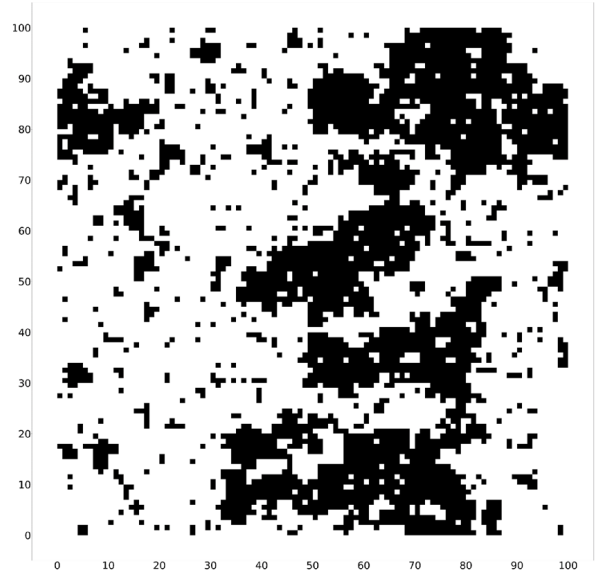
KbT=3.000000 2



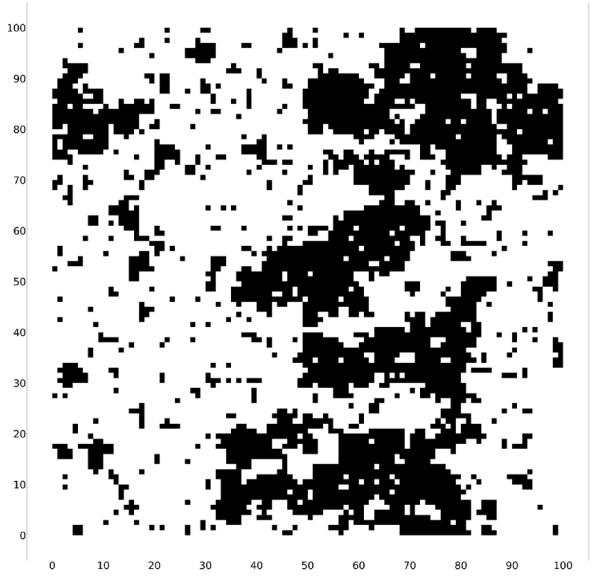
KbT=3.000000 3



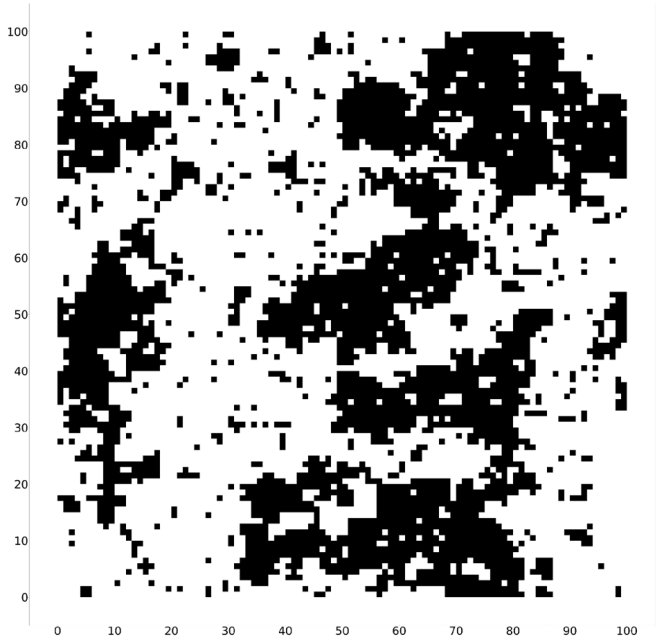
KbT=3.000000 4



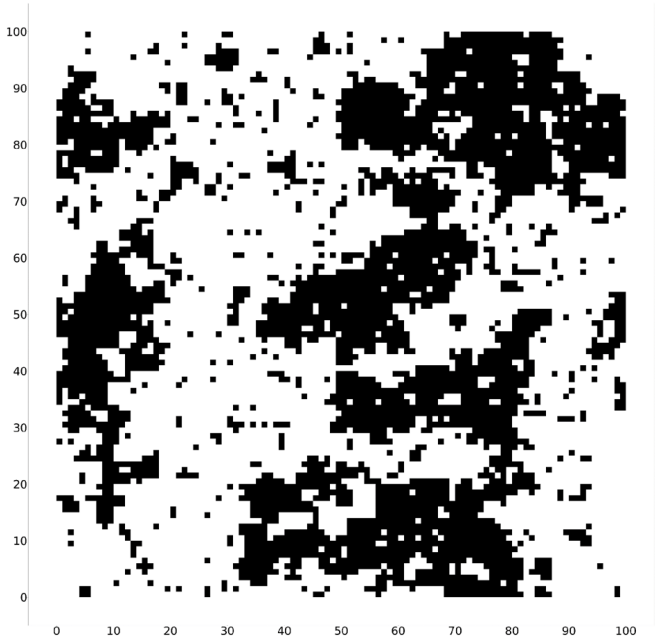
KbT=3.000000 5



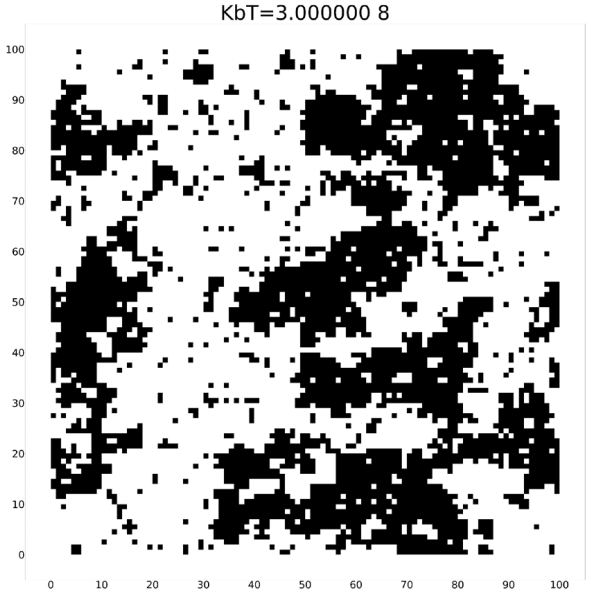
KbT=3.000000 6



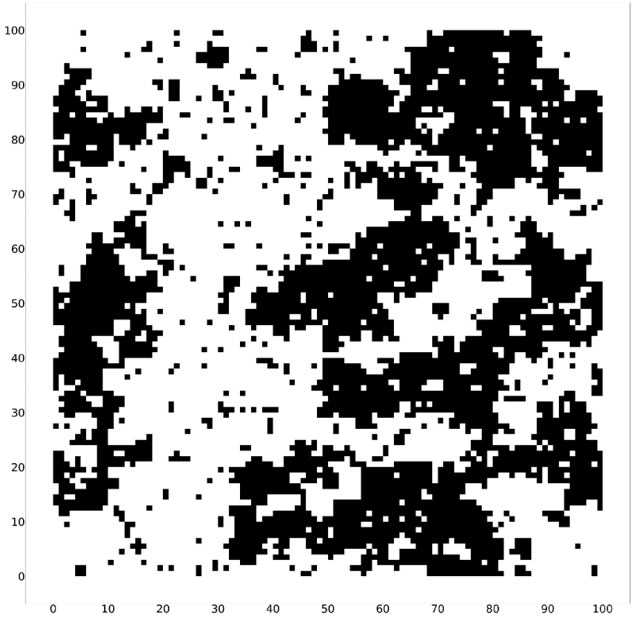
KbT=3.000000 7



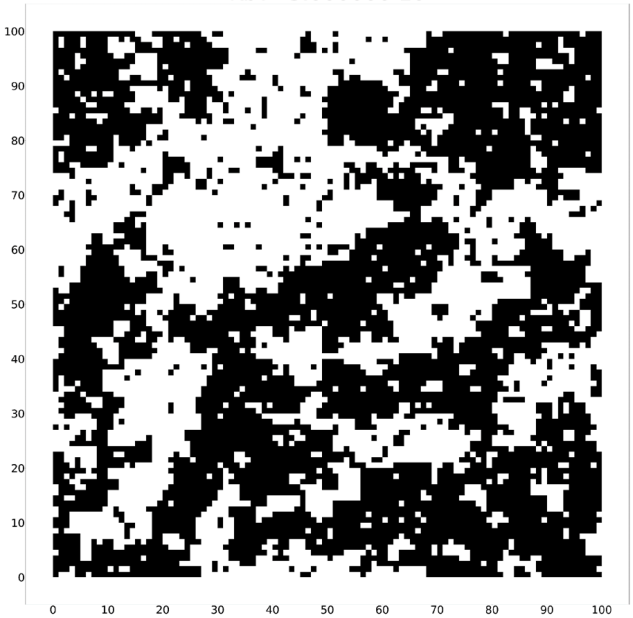
KbT=3.000000 8



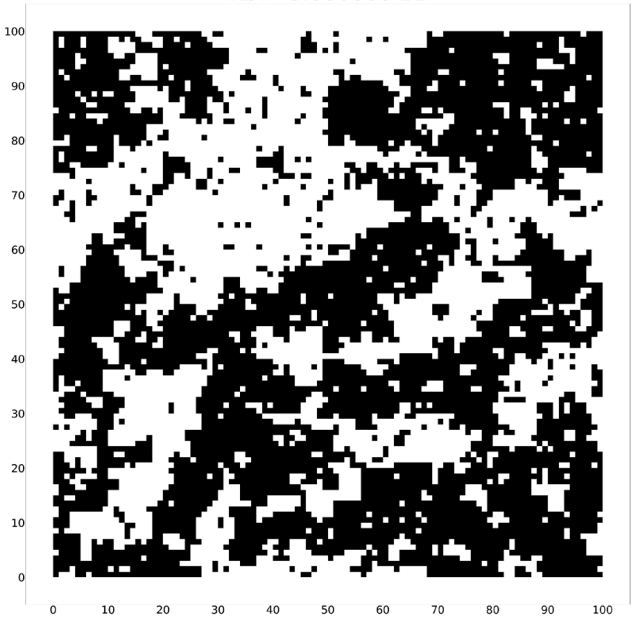
KbT=3.000000 9

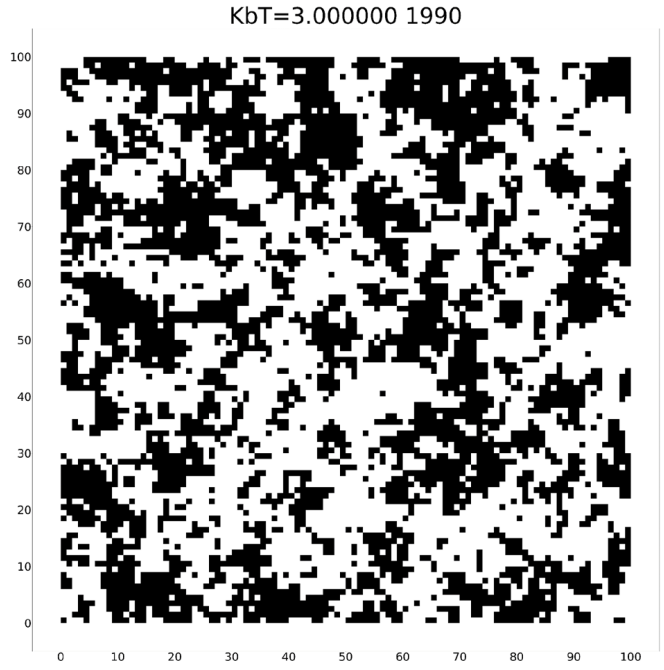


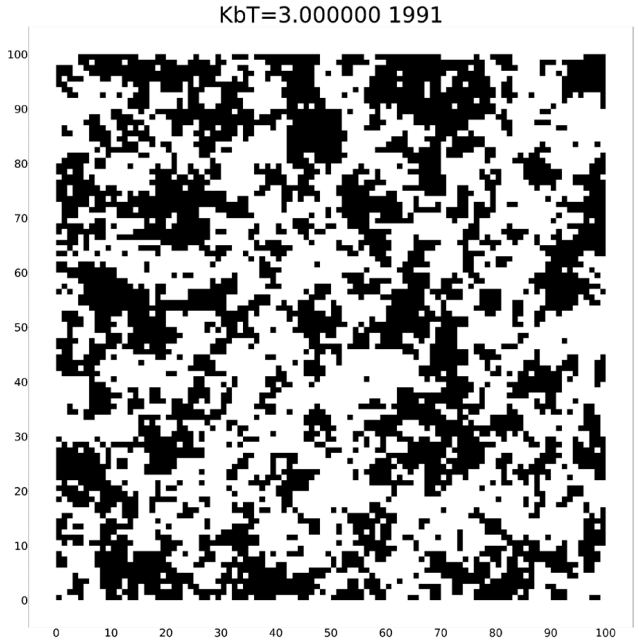
KbT=3.000000 10

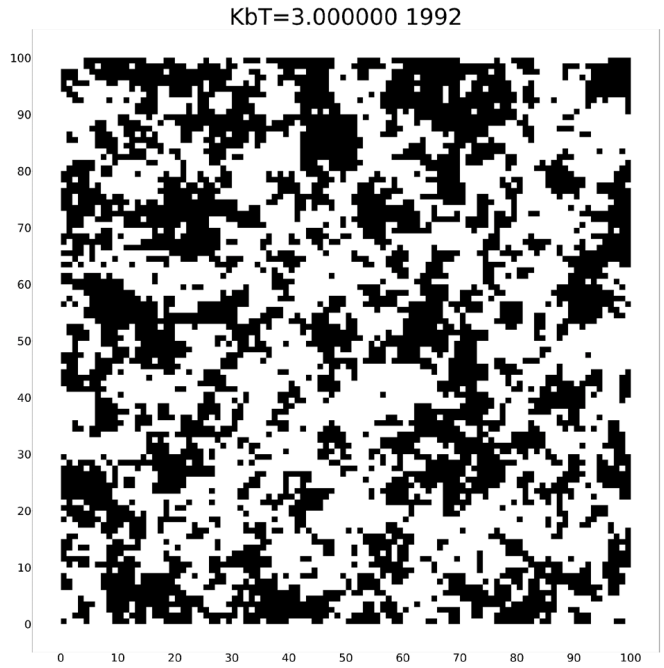


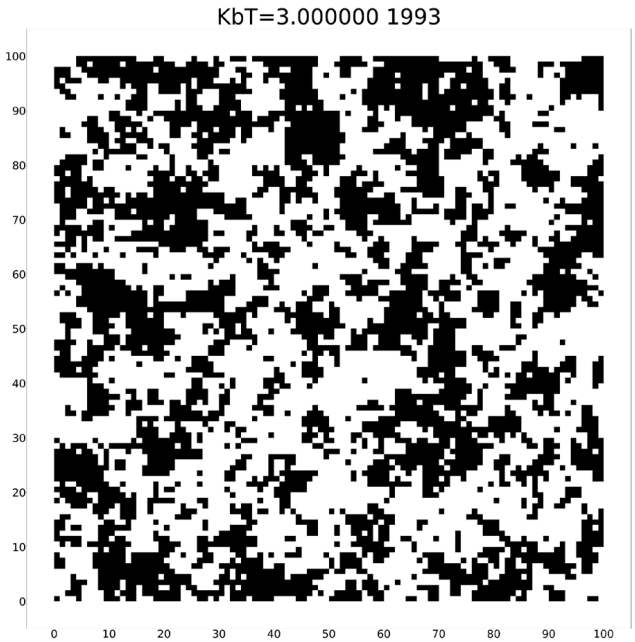
KbT=3.000000 11

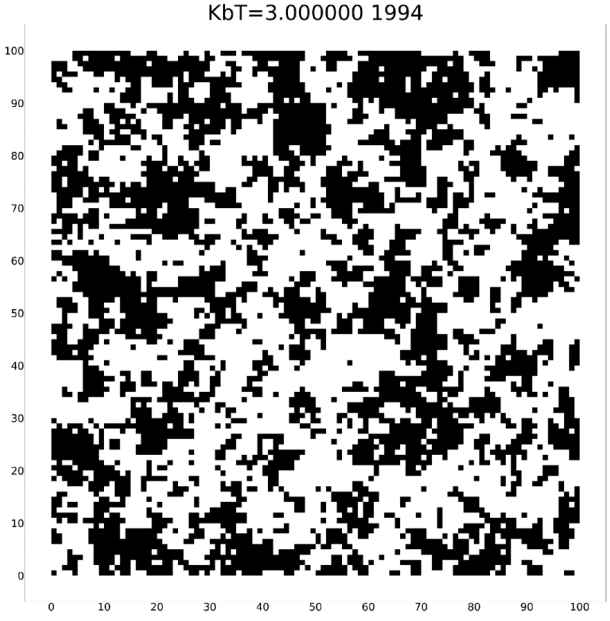


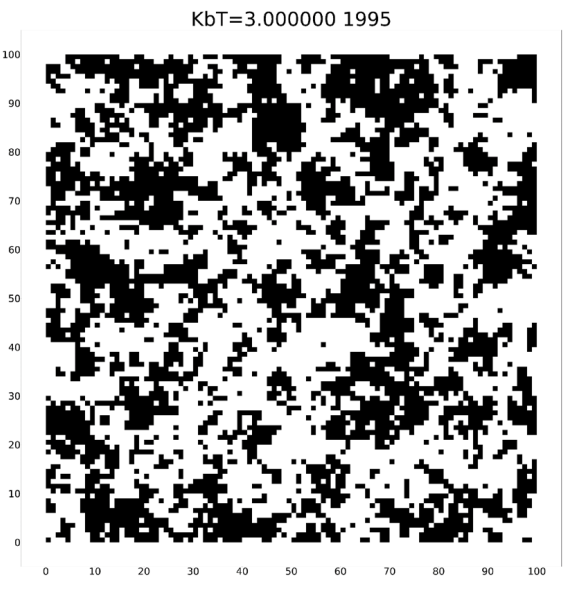

KbT=3.000000 1990


KbT=3.000000 1991


KbT=3.000000 1992


KbT=3.000000 1993


KbT=3.000000 1994


KbT=3.000000 1995

4 Summary

We could know the theoretical value of the critical temperature is 2.269 for the case J=1, H=0(our parameters) from literature [7]. Our result from SSF is 2.1025, with an error of 7.3 percent, while the result from Wolff is 2.53, with an error of 11.5 percent. Both of them are tolerated while Wolff would have much faster speed.

References

[1] Giovanni Gallavotti. *Statistical mechanics: A short treatise*. Springer Science & Business Media, 1999.

[2] Nick Metropolis. The beginning of the monte carlo method.

[3] Barry A Cipra. An introduction to the ising model. *The American Mathematical Monthly*, 94(10):937–959, 1987.

[4] Louis A Turner. Zeroth law of thermodynamics. *American Journal of Physics*, 29(2):71–76, 1961.

[5] Ulli Wolff. Collective monte carlo updating for spin systems. *Phys. Rev. Lett.*, 62:361–364, Jan 1989.

[6] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Third edition, September 2011.

[7] KS Soldatov, KV Nefedev, V Yu Kapitan, and PD Andriushchenko. Approaches to numerical solution of 2d ising model. In *Journal of Physics: Conference Series*, volume 741, page 012199. IOP Publishing, 2016.

Acknowledgement

When I was an undergraduate in Wuhan University, more than one professors and senior postgraduates have told me, just choose a direction to do research. Although I have chosen astronomy to carry out a small research since it is one of the directions interests me most, I still want to get to know different directions because I don't want to make mistakes in choosing directions, just like I chose Computer Science as major in undergraduate but actually don't have much interest.

So I chose to come to CUHK to have more understanding about different directions. As a result, when I heard PHYS4061 is carried out in many Solid Physics background, I was very happy because I have not touched this direction before. This course thus could not only let me know the special computational methods in physics, but could also be regarded as a complement to my physics knowledge-Solid State Physics is a fundamental Physics course for Physics Major, but due to the limited time in WHU, I have not learnt this course.

During the 4 well designed labs of Project A, I have paid a lot of time but also learnt a lot. I have been trained to study by myself(such as reading Prof. Hui Pak Ming's notes about solid state physics) to cover the gap in Solid State Physics, which is a precious experience since in further studies self learning is more and more important.

During the Project B and, I have learnt a lot too, not only in physics but also in Computer Science. For example just like the part FIFO working better for cluster growing mechanism: we could give a rough proof, it is the first time I have found the knowledge what I learnt in Data Structure in my undergraduate really could have some use.

Many thanks to Prof. Zhu to design such a good course and his teaching, many thanks to Wenjing and Tommy for their careful help as TAs. Without you my studies in CUHK could not be so happy and enriching.

Thanks again! Best wishes to all of you!