Macalester Journal of Physics and Astronomy

Volume 3
Issue 1 *Spring 2015*

Article 9

May 2015

Simulating Correlated Disorder in Spin Glass

Jared D. Willard
Macalester College, jwillard@macalester.edu

Follow this and additional works at: https://digitalcommons.macalester.edu/mjpa

Part of the Astrophysics and Astronomy Commons, and the Physics Commons

Recommended Citation

Willard, Jared D. (2015) "Simulating Correlated Disorder in Spin Glass," *Macalester Journal of Physics and Astronomy*: Vol. 3: Iss. 1, Article 9.
Available at: https://digitalcommons.macalester.edu/mjpa/vol3/iss1/9

This Capstone is brought to you for free and open access by the Physics and Astronomy Department at DigitalCommons@Macalester College. It has been accepted for inclusion in Macalester Journal of Physics and Astronomy by an authorized editor of DigitalCommons@Macalester College. For more information, please contact scholarpub@macalester.edu.

Macalester College

Simulating Correlated Disorder in Spin Glass

Abstract

Almost all materials undergo spontaneous symmetry breaking at sufficiently low temperatures. For most magnetic materials, the spin rotational symmetry is broken to form magnetic ordering. The discovery of metallic alloys which fail to form conventional magnetic order has remained a puzzle for the last few decades. Unfortunately, analytical calculations cannot provide an unbiased answer for the problem. Furthermore, on the numerical side, Monte Carlo simulations require extremely long equilibration times. The parallel tempering method has proven a powerful tool to alleviate the long equilibration time. With the extensive efforts of numerical simulation research, some of the idealized models have been studied in detail. The general consensus is that for models with uncorrelated disorder there exists a finite spin glass critical temperature in three dimensions. However, it is not hard to imagine that, in real materials, the disorder is somewhat correlated, meaning the correlation exists between a completely random distribution and the correlation you would see in a crystalline lattice. In this work, we employ modern spin glass simulation techniques to study a prototype spin glass model with correlated disorder. We find that the critical temperature is enhanced due to the correlated disorder.

Keywords

spin glass condensed matter physics

This capstone is available in Macalester Journal of Physics and Astronomy: https://digitalcommons.macalester.edu/mjpa/vol3/iss1/9

Simulating Correlated Disorder in Spin Glass

Jared Willard

Macalester College

Louisiana State University Department of Physics and Astronomy

Louisiana Alliance for Simulation Guided Materials Applications

May 12, 2015

1 Abstract

Almost all materials undergo spontaneous symmetry breaking at sufficiently low temperatures. For most magnetic materials, the spin rotational symmetry is broken to form magnetic ordering. The discovery of metallic alloys which fail to form conventional magnetic order has remained a puzzle for the last few decades. Unfortunately, analytical calculations cannot provide an unbiased answer for the problem. Furthermore, on the numerical side, Monte Carlo simulations require extremely long equilibration times. The parallel tempering method has proven a powerful tool to alleviate the long equilibration time. With the extensive efforts of numerical simulation research, some of the idealized models have been studied in detail. The general consensus is that for models with uncorrelated disorder there exists a finite spin glass critical temperature in three dimensions. However, it is not hard to imagine that, in real materials, the disorder is somewhat correlated, meaning the correlation exists between a completely random distribution and the correlation you would see in a crystalline lattice. In this work, we employ modern spin glass simulation techniques to study a prototype spin glass model with correlated disorder. We find that the critical temperature is enhanced due to the correlated disorder.

2 Introduction

Spin glasses are magnetic systems in which the magnetic properties of the system are frustrated due to frozen-in structural disorder. The concept of geometrical frustration is ubiquitous in complexity theory and the concept of degeneracy in physics.

The frustrated interactions exist when a certain spin doesn't have a clear route to the lowest energy distribution and cannot be predicted to go any certain route as states with equal energy continuously change in the simulation, or in experimental procedures. Due to this disorder, no conventional ordering such as ferromagnetism and antiferromagnetism can be established. In these materials, The term "glass" illustrates the parallel between the *magnetic* disorder of the spin glass and the *positional* disorder of chemical glass, such as window class, where the atomic bond structure exhibits no long range ordering and is highly irregular. The physics of spin glasses spans several fundamental principles and investigations in statistical physics, and the analysis of the empirical evidence has significant multidisciplinary character —these two reasons motivated spin glass research across the past few decades and have continued to present day. Emerging research in spin glass primarily use the Edwards-Anderson model, or the Infinite Range Model and the model of Sherrington and Kirkpatrick, which are both rooted in mean-field theory. The specific numerical approach we choose to simulate and study spin glasses in this paper is based on a variation of the numerical Edwards-Anderson model, and are known as the parallel tempering Monte Carlo and the Metropolis algorithm.

One of the main problems with simulating spin-glass, and finding where the system has a first-order phase transition in which magnetic disorder is *frozen* into the material, is that any given spin glass system when looked at through time gets "stuck" inside metastable states that are not the lowest energy state (ferromagnetic ordering, all spins aligned) due to the high amount of geometrical complexity. This underlying notion

of complexity is incredibly applicable to many areas of computational science with metastable states, like artificial neural networks and theoretical brain research. The parallel tempering Monte Carlo aims to surpass these boundaries, and explore spin configurations at energy levels that would not normally be realized due to this concept of metastability.

In the Edwards-Anderson model, the energy levels are calculated using the Hamiltonian in Equation 1. Here, s_i and s_j represent nearest neighbor spins and J_{ij} is the

$$H = -\Sigma_{ij} J_{ij} s_i s_j \tag{1}$$

coupling constant traditionally allocated the values 1 and -1 for spin up and spin down, and are randomly given a value based on a Gaussian distribution. This has been fruitful in the past because the problem is solvable numerically for a transition temperature T_c. However, it is not unrealistic to think that spins will be randomly distributed according to a Gaussian distribution, and instead we investigate a weakly correlated system where J_{ij} is given by Equation 2. Here n and ζ are identically distributed according to a bimodal

$$J_{ij} = n_i \zeta_j + \zeta_i n_j \tag{2}$$

distribution, applying a weakly correlated disorder that aims to be more practical for simulating real systems. Furthering the realistic system, to obtain the physical observables (magnetization, magnetic susceptibility, spin glass correlation length, etc) we perform a trace over the partition function in Equation 3. The calculation of the

$$Z = \Sigma_{\{s\}} exp[-\beta E(\{s\})] \tag{3}$$

partition function involves a summation over all possible spin configurations, with β representing inverse temperature and $E(\{s\})$ representing the energy calculation of a

given spin configuration *s*. However, the number of spin configurations grows as 2^N, which is not feasible for even moderately large numbers of spins. For this reason, using the Monte Carlo method to sample stochastically becomes even more significant in reducing computation times, as well as surpassing the obstacle of metastability.

3 Methodology

3.1 Monte Carlo Simulations

The motivation for Monte Carlo integration is rooted in the fact that most traditional integration methodologies fail for integrals with a large number of inputs. At the same time, the space dimension of the phase space of classical spin glass systems is immense. Specifically, the phase space dimension for N classical particles in three spatial dimensions is $6N$ (three spatial coordinates and three momentum components in the wave vector). This is infeasible for the case of N classical Edwards-Anderson spins which can take the coupling constants as given in Equation 2. In this case the phase space dimension is 2^N , since we're summing over all possible spin configurations in Equation 3, which turns out to be infeasible to simulate numerically even with only mediocre amounts of spins. Therefore, integration schemes such as Monte Carlo methods, where the error is independent of the space dimension, are needed.

3.2 Metropolis Algorithm

Using the previously mentioned principles behind the Monte Carlo simulation, we can apply these methods to statistical physics to sample the average of an observable in a given system. However, to select system states as the system moves through time, we must somehow encode information about the system in the transition probabilities from state to state. Since the Boltzmann distribution contains all the information we need about the system, a method must be able to sample from the Boltzmann distribution. This method is known as the Metropolis Algorithm. The Metropolis algorithm generates a Markov chain, a type of Monte Carlo method, of successive states. The new state is generated from the previous state based on a specific transition probability given by the equilibrium Boltzmann distribution. Figure 1 shows brief pseudocode for the process.

3.3 Parallel Tempering Monte Carlo

Throughout the history of spin glass research, scientists have continually encountered the problem of undesirable local minima. In the free energy landscape of the state space of the many-body system with frustrated interaction, many local minima exist that are separated by large energy barriers. Consequently, tunneling through these barriers is extremely unlikely due to the calculated probabilities from the Boltzmann distribution, and therefore the relaxation times toward equilibrium in a given system tend to be extremely long.

A method to get around this is known as the parallel tempering Monte Carlo method. This method looks to overcome these “mountains” in the energy landscape by running several replicas of the system of interest at varying temperatures. Therefore, as with most systems in statistical physics, higher temperatures allows for the escape of

these metastable states in local minima before relaxing to lower temperatures. In experiment, we see a several orders of magnitude speedup (Katzgrabber 2011). The temperature space is navigated in Monte Carlo fashion with two copies of the system at neighboring temperatures being swapped based on the magnitudes of the temperature difference and the energy of the system calculated by the Hamiltonian in Equation 1 after a given number of Monte Carlo lattice sweeps. This instantiates a random walk through temperature space for each replica of the system.

4 Results

Simulations were ran on the supercomputer SuperMike2 at the Louisiana State University High Performance Computing Institute. After 150000 Monte Carlo lattice sweeps with system replicas being swapped every 100 sweeps, and energy levels being measured every 10 sweeps (code in Appendix A1), we measure correlation length, a dimensionless quantity shown in Equation 4, to look for phase changes. This is demonstrated in Figure 2, showing correlation length as a function of inverse temperature exhibiting a phase transition at $\beta=0.37$. We also plot the susceptibility ratios for system sizes $L=4,6,8,10$ (dimension of the lattice cube), and find the larger three sizes cross at $\beta=0.31$, also showing a finite critical temperature.

5 Discussion

Phase transitions are seen by examining certain dimensionless quantities that scale at the critical point for different system sizes. Correlation length as measured in Figure 2 is calculated as follows in Equation 4. Here L is the dimension of the lattice cube, and

χ(k) is the magnetic susceptibility of the system for a given section of the wave vector *k.* The phase transition we find at β=0.37 in

$$\frac{\xi}{L} = 1/\left(2\sin\left(\frac{2\pi}{L}\right)\right)\sqrt{\frac{\chi(\mathbf{k_0})}{\chi(\mathbf{k_1})} - 1} \tag{4}$$

Figure 2 is a slightly higher temperature than we would have seen if it had been a Gaussian distribution instead of a weakly correlated distribution (Binder 1986) We also plot the susceptibility ratios for system sizes L=4,6,8,10 as calculated in Equation 5 and Equation 6, and find the larger three sizes cross at β=0.31,

$$\chi(\mathbf{k}) = \frac{1}{N}\sum_{i,j}\left[< s_i s_j >_T^2\right]\exp\left(i\boldsymbol{k}\left(\boldsymbol{r_i} - \boldsymbol{r_j}\right)\right) \tag{5}$$

$$R_{12} = \frac{\chi(\boldsymbol{k_1})}{\chi(\boldsymbol{k_2})} \tag{6}$$

also showing a finite critical temperature higher than a Gaussian distribution (Binder 1986). Here, *N* is the number of spins and *r* is the position vector for a given spin in the lattice cube. At these finite critical temperatures, we see measurements diverge at the critical point, showing the spin glass characterized magnetic ordering.

6 Conclusion

In this paper we presented a successful implementation of the parallel tempering Monte Carlo method using the Metropolis algorithm and OpenMPI for parallelization on a high performance computing system. On a weakly correlated cubic lattice Edwards-Anderson model we find a spin glass transition at a critical temperature slightly higher than thought of the Gaussian distribution previously found.

In future work, we would have examined the error bars behind the simulation to see if the results are valid and if they scale to larger system sizes. Also, other geometries in addition to the cubic lattice, such as a diamond lattice or triangular lattice, may also yield interesting results.

7 Acknowledgements

This project would not have been possible without the guidance of Dr. Ka Ming Tam, and graduate students Sheng Feng and Yi Fang of the LSU Computational Condensed Matter Physics research group.

9 Figures

```
1  algorithm ising_metropolis(T,steps)
2      initialize starting configuration S
3      initialize O = 0
4
5      for(counter = 1 ... steps) do
6          generate trial state S'
7          compute p(S -> S',T)
8          x = rand(0,1)
9          if(p > x) then
10             accept S'
11         fi
12
13         O += O(S')
14     done
15
16 return  O/steps
```

Figure 1

Psuedocode illustrating the fundamental idea behind the Metropolis algorithm (Katzgrabber 2011). In this case “O” represents any observable of the system.

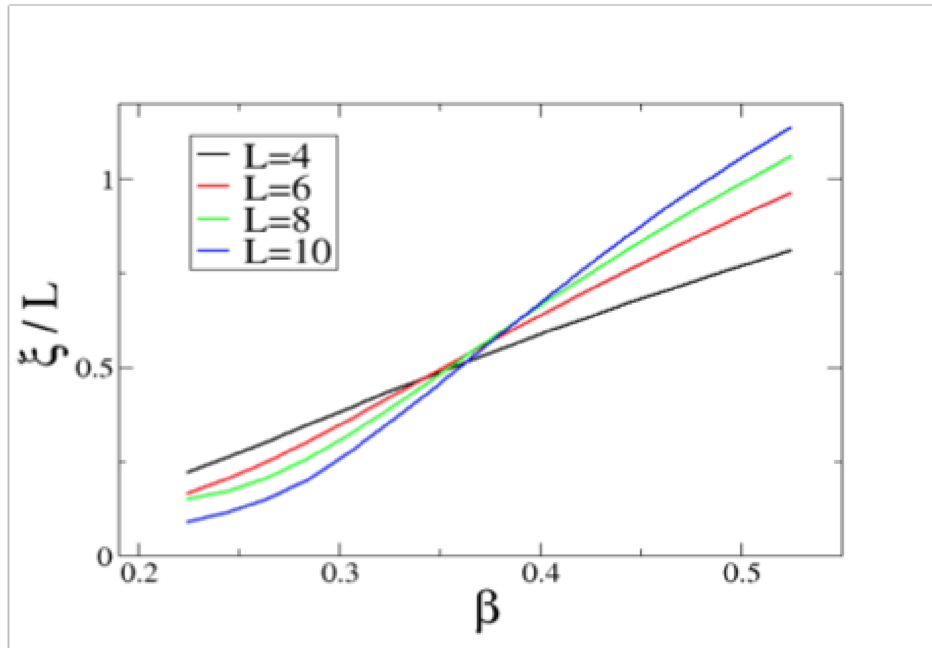

Figure 2

Graph showing correlation length as a function of inverse temperature

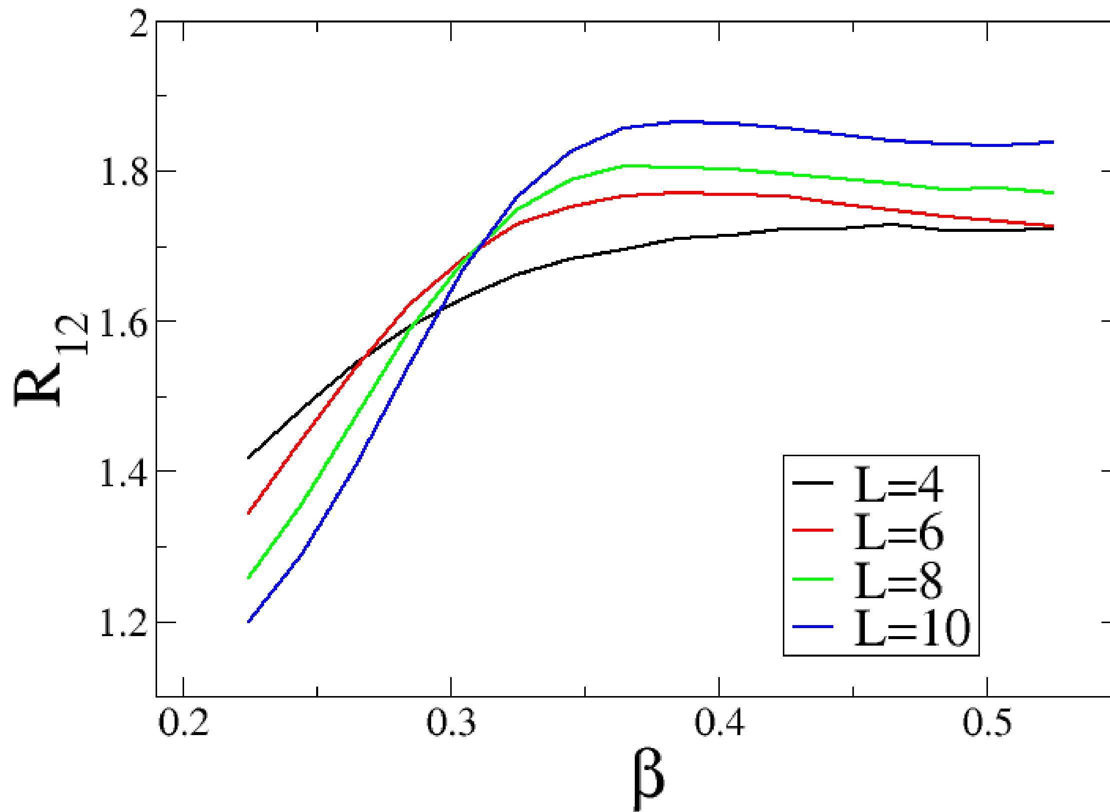

Figure 3

Graph showing susceptibility ratios of two wave vector moments as a function of inverse temperature.

Appendix A1 - Simulation Code (C++)

```
#include <iostream>

#include <string>

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <mpi.h>
```

```
//#include <gsl_math.h>

#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
float ran2(long *idum)
{
int j;
long k;
static long idum2=123456789;
static long iy=0;
static long iv[NTAB];
float temp;
if (*idum <= 0) {
if (-(*idum) < 1) *idum=1;
```

```
else *idum = -(*idum);
idum2=(*idum);
for (j=NTAB+7;j>=0;j--) {
k=(*idum)/IQ1;
*idum=IA1*(*idum-k*IQ1)-k*IR1;
if (*idum < 0) *idum += IM1;
if (j < NTAB) iv[j] = *idum;
}
iy=iv[0];
}
k=(*idum)/IQ1;
*idum=IA1*(*idum-k*IQ1)-k*IR1;
if (*idum < 0) *idum += IM1;
k=idum2/IQ2;
idum2=IA2*(idum2-k*IQ2)-k*IR2;
if (idum2 < 0) idum2 += IM2;
j=iy/NDIV;
iy=iv[j]-idum2;
iv[j] = *idum;
if (iy < 1) iy += IMM1;
if ((temp=AM*iy) > RNMX) return RNMX;
else return temp;
}



FILE* f1;
```

```
const int Lx = 8;

const int Ly = 8;

const int Lz = 8;

const float J0 = 0.0;

const float pi = 3.14159265359;

const float eta = 0.5;

const float hac = 0.0;

const float period = 1280;

const int TotTime = 150000;


const int Nr = 600;


int NS = Lx*Ly*Lz;


float Jij[6*Lx*Ly*Lz];

int nnlist[6*Lx*Ly*Lz];

int cor[3*Lx*Ly*Lz];

int s[Lx*Ly*Lz];


int T;


long seed;
```

```
void get_Jij()

{

// float r = (float)rand()/(float)RAND_MAX;


float rset1[NS], rset2[NS];




for(int i =0; i < NS; i++)

{

        float r1 = ran2(&seed);

        float r2 = ran2(&seed);

        float gr1 = sqrtf(-2.0*logf(r1)) * cosf(2.0*pi*r2);

        float gr2 = sqrtf(-2.0*logf(r1)) * sinf(2.0*pi*r2);

  rset1[i] = gr1 + J0;

}


for(int i =0; i < NS; i++)

{

        float r1 = ran2(&seed);

        float r2 = ran2(&seed);

        float gr1 = sqrtf(-2.0*logf(r1)) * cosf(2.0*pi*r2);

        float gr2 = sqrtf(-2.0*logf(r1)) * sinf(2.0*pi*r2);

        rset2[i] = gr1 + J0;

}



 for(int i=0; i < Lx*Ly*Lz; i++)
```

```
{
        for(int j=0; j<3; j++)
  {
        Jij[6*i + j] = -rset1[i] * rset2[nnlist[6*i+j]] - rset2[i] * rset1[nnlist[6*i+j]];
        Jij[6*nnlist[6*i+j]+(j+3)] = -rset1[i] * rset2[nnlist[6*i+j]] - rset2[i] * rset1[nnlist[6*i+j]];
}

}

/* for(int i = 0; i < Lx*Ly*Lz; i++)
{

printf("before, Jij, %d, %f, %f, %f, %f, %f, %f \n", i, Jij[6*i],Jij[6*i+1],Jij[6*i+2],Jij[6*i+3],Jij[6*i+4],Jij[6*i+5]);
printf(" i,  NN, %d, %f, %f, %f, %f, %f, %f \n", i,
Jij[6*nnlist[6*i]+3],Jij[6*nnlist[6*i+1]+4],Jij[6*nnlist[6*i+2]+5],Jij[6*nnlist[6*i+3]+0],Jij[6*nnlist[6*i+4]+1],Jij[6*nnli
st[6*i+5]+2]);

}
*/
}

/*
float Ri[3], Rj[3];
float Jijsave[6*Lx*Ly*Lz];
```

```
for(int i = 0; i < 6 * Lx*Ly*Lz; i++)

{

Jijsave[i] = Jij[i];

}



 for(int i1=0; i1 < Lx*Ly*Lz; i1++)

{

            for(int i2=0; i2< 3 ; i2++)

  {


            float temp1 = 0.0;

            float temp2 = 0.0;


            Ri[0] = 1.0*cor[3*i1];

            Rj[1] = 1.0*cor[3*j1+1];

            Rj[2] = 1.0*cor[3*j1+2];

            if(j2 == 0) Rj[0] = Rj[0] + 0.5;

            if(j2 == 1) Rj[1] = Rj[1] + 0.5;

            if(j2 == 2) Rj[2] = Rj[2] + 0.5;


            float Rij2 = (Ri[0]-Rj[0])*(Ri[0]-Rj[0]) + (Ri[1]-Rj[1])*(Ri[1]-Rj[1]) + (Ri[2]-Rj[2])*(Ri[2]-Rj[2]);


            temp1 = temp1 + expf(-Rij2/(eta*eta));

            temp2 = temp2 + Jijsave[6*j1+j2]*expf(-Rij2/(2.0*eta*eta));

  }

}
```

```
 // Jij[6*i1+i2] = temp2 / sqrtf(temp1);
            Jij[6*nnlist[6*i1+i2]+(i2+3)] = temp2 / sqrtf(temp1);
            Jij[6*i1+i2] = Jij[6*nnlist[6*i1+i2]+(i2+3)] ;



}
}



// make correlated randomness


/*
 for(int i = 0; i < Lx*Ly*Lz; i++)
{


printf(" i, Jij, %d, %f, %f, %f, %f, %f, %f \n", i, Jij[6*i],Jij[6*i+1],Jij[6*i+2],Jij[6*i+3],Jij[6*i+4],Jij[6*i+5]);
printf(" i,  NN, %d, %f, %f, %f, %f, %f, %f \n", i,
Jij[6*nnlist[6*i]+3],Jij[6*nnlist[6*i+1]+4],Jij[6*nnlist[6*i+2]+5],Jij[6*nnlist[6*i+3]+0],Jij[6*nnlist[6*i+4]+1],Jij[6*nnli
st[6*i+5]+2]);





}


 return;
}
```

```
*/

float mag ()
{
float m = 0.0;

for(int i=0; i < Lx*Ly*Lz; i++)
{
 m += (float)s[i];
}

m = m / (Lx*Ly*Lz);

return m;
}


void chi(float SiSj[NS*NS])
{


for(int i=0; i < NS; i++)
{
for(int j=0; j < NS; j++)
{
```

```
 SiSj[i+NS*j] = (float)(s[i]*s[j]) ;
}
}



}




void lattice()
{
int ix1, iy1, iz1;
int ix2, iy2, iz2;
int ix, iy, iz;
int site;
int counter;

for(site=0; site < Lx*Ly*Lz; site++)
{

iz = (site) / (Lx*Ly) ;

iy = (site - iz*Lx*Ly ) / Lx ;

ix = (site - (iz)*Lx*Ly - (iy)*Lx)  ;

ix1 = ix - 1;
```

```
if(ix1 == -1) ix1 = Lx - 1;
ix2 = ix + 1;
if(ix2 == Lx) ix2 = 0;

iy1 = iy - 1;
if(iy1 == -1) iy1 = Ly - 1;
iy2 = iy + 1;
if(iy2 == Ly) iy2 = 0;

iz1 = iz - 1;
if(iz1 == -1) iz1 = Lz - 1;
iz2 = iz + 1;
if(iz2 == Lz) iz2 = 0;

int nnsite[6];

cor[3*site+0] = ix;
cor[3*site+1] = iy;
cor[3*site+2] = iz;

nnsite[2] = (iz1)*Lx*Ly+(iy)*Lx+ix;
nnsite[5] = (iz2)*Lx*Ly+(iy)*Lx+ix;

nnsite[1] = (iz)*Lx*Ly+(iy1)*Lx+ix;
nnsite[4] = (iz)*Lx*Ly+(iy2)*Lx+ix;

nnsite[0] = (iz)*Lx*Ly+(iy)*Lx+ix1;
```

```
nnsite[3] = (iz)*Lx*Ly+(iy)*Lx+ix2;

counter = site*6;
nnlist[counter] = nnsite[0];
counter = counter + 1;
nnlist[counter] = nnsite[1];
counter = counter + 1;
nnlist[counter] = nnsite[2];
counter = counter + 1;
nnlist[counter] = nnsite[3];
counter = counter + 1;
nnlist[counter] = nnsite[4];
counter = counter + 1;
nnlist[counter] = nnsite[5];
/*
printf(" x,y,z,counter, %6d,%6d,%6d,%6d \n", ix,iy,iz,counter);
printf(" %6d,%6d,%6d,%6d,%6d,%6d \n", ix1,ix2,iy1,iy2,iz1,iz2);
printf(" %6d,%6d,%6d,%6d,%6d,%6d \n", nnsite[0],nnsite[1],nnsite[2],nnsite[3],nnsite[4],nnsite[5]);
*/

}


return;

}
```

```
float Etot ()
{
int ix1, iy1, iz1;
int ix, iy, iz;
int site;
float energy;
int counter;

energy = 0.0;

for(site=0; site < Lx*Ly*Lz; site++)
{

int nnsite[3];
counter = site * 6;
nnsite[0] = nnlist[counter];
counter = counter + 1;
nnsite[1] = nnlist[counter];
counter = counter + 1;
nnsite[2] = nnlist[counter];


for(int i = 0; i <= 2 ; i++)
  {energy = energy + Jij[6*site+i]* (float)s[nnsite[i]]*s[site];
```

```
//          printf(" field %6d, %f \n", i, field );

            }

//          printf(" field %f \n", field );

//printf(" %6d,%6d,%6d,%6d,%6d,%6d,%6d \n", site, nnsite[0], nnsite[1],
nnsite[2],nnsite[3],nnsite[4],nnsite[5] );

}

//printf(" energy %f \n", energy);

return energy;
}

float dE (int site)
{
int ix1, ix2, iy1, iy2, iz1, iz2;

int nnsite[6];
```

```
nnsite[0] = nnlist[6*site + 0];
nnsite[1] = nnlist[6*site + 1];
nnsite[2] = nnlist[6*site + 2];
nnsite[3] = nnlist[6*site + 3];
nnsite[4] = nnlist[6*site + 4];
nnsite[5] = nnlist[6*site + 5];

float field;
//printf(" %6d,%6d,%6d,%6d,%6d,%6d,%6d \n", site, nnsite[0], nnsite[1],
nnsite[2],nnsite[3],nnsite[4],nnsite[5] );

field = Jij[6*site+0]*s[nnsite[0]] + Jij[6*site+1]*s[nnsite[1]] + Jij[6*site+2]*s[nnsite[2]] ;
field = field  + Jij[6*site+3]*s[nnsite[3]] + Jij[6*site+4]*s[nnsite[4]] + Jij[6*site+5]*s[nnsite[5]];
field = -2.0 * field * s[site];

return field;
}

void initial()
{
```

```
int i, j, k;



for(i = 0; i < Lx ; i++)

  {

        for(j = 0; j < Ly; j++)

        {

        for(k = 0; k < Lz; k++)

        {

        float r = (float)rand()/(float)RAND_MAX;

        if(r  < 0.5)

        {

        s[i*Lx*Ly+j*Lx+k] = 1;

        }

        else

        {

        s[i*Lx*Ly+j*Lx+k] = -1;

        }

        }

        }

        }

}




/* main progra */
```

```
int main(int argc, char *argv[])
{
        int my_id, nprocs;

        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&my_id);

        MPI_Status status;
/*
const int Lx = 16 ;
const int Ly = 16 ;
const int Lz = 16 ;
*/

//  srand (my_id+time(NULL))
//

        seed = -1 * ( my_id + time(NULL));




//printf("my_id, seed, %d, %ld, \n", my_id, seed );
//sleep(100);

char fileout[20];
sprintf(fileout, "%d_%d.dat", Lx, my_id);
```

```
f1 = fopen(fileout,"wt");





float SiSj2_Rave[NS*NS];
float SiSj_Rave[NS*NS];
float SiSjave[NS*NS];
float mave;
float mave2;
float mave4;
float eave;
float eave2;
float chippave;


int counter;

float beta;
int i, j, k;


for(i=0;i<NS*NS;i++)
{
 SiSj_Rave[i] = 0.0;
 SiSj2_Rave[i] = 0.0;
```

```
}

lattice();

for(int ir = 0; ir < Nr; ir++)
{

if(my_id == 0)
{
get_Jij();
}

MPI_Bcast(&Jij,6*Lx*Ly*Lz,MPI_FLOAT,0,MPI_COMM_WORLD);

beta = 0.25 + 0.034375 *  my_id ;

initial();

for(int l = 0; l < NS*NS; l++) SiSjave[l] = 0.0;
mave4 = 0.0;
eave = 0.0;
eave2 = 0.0;
```

```
chippave = 0.0;

counter = 0;


//loop over T

for(T = 1; T < TotTime; T++)

{







//printf("my_id, T, %d,%d, \n", my_id,T);


//float ht = hac * sin(T*2.0*pi/period);

// loop over all sites


for(int site = 0; site < Lx*Ly*Lz ; site++)

  {

        float temp = dE(site);

        float prob = temp * beta;

//      printf(" prob %f \n", prob );


        prob = expf(-prob);

        float r = ran2(&seed);

//      float r = (float)rand()/(float)RAND_MAX;

//      printf(" site %6d \n", site );

//      printf(" r prob %f, %f \n", r, prob );
```

```
//          printf(" prob %f \n", prob );

//fast no-if statement
            s[site] = s[site] * nearbyintf(copysignf(1.0,r-prob));

/*
            if(prob >= r)
            {
            s[site] = -s[site];
//          printf(" site updated %6d \n", site );

            }

*/

            }




//every 10 steps!!!
if(T%10 == 0 && T%20 !=0)
//if(T%10 == 0 )
{

//printf("in 10 & !20 loop, %d, \n", my_id);
```

```
float energy;
energy = Etot();
//float r = (float)rand()/(float)RAND_MAX;
float r = ran2(&seed);

float Renergy;
float Rr;
int   Rs[Lx*Ly*Lz];
float Rbeta;
if(my_id%2 == 0)
{
  MPI_Send(&energy,1,MPI_FLOAT,my_id+1,1,MPI_COMM_WORLD);
  MPI_Send(&r,1,MPI_FLOAT,my_id+1,2,MPI_COMM_WORLD);
  MPI_Send(&beta,1,MPI_FLOAT,my_id+1,3,MPI_COMM_WORLD);
  MPI_Send(&s,Lx*Ly*Lz,MPI_INT,my_id+1,4,MPI_COMM_WORLD);

  MPI_Recv(&Renergy,1,MPI_FLOAT,my_id+1,5,MPI_COMM_WORLD,&status);
  MPI_Recv(&Rr,1,MPI_FLOAT,my_id+1,6,MPI_COMM_WORLD,&status);
  MPI_Recv(&Rbeta,1,MPI_FLOAT,my_id+1,7,MPI_COMM_WORLD,&status);
  MPI_Recv(&Rs,Lx*Ly*Lz,MPI_INT,my_id+1,8,MPI_COMM_WORLD,&status);
}
else
{
  MPI_Recv(&Renergy,1,MPI_FLOAT,my_id-1,1,MPI_COMM_WORLD,&status);
  MPI_Recv(&Rr,1,MPI_FLOAT,my_id-1,2,MPI_COMM_WORLD,&status);
  MPI_Recv(&Rbeta,1,MPI_FLOAT,my_id-1,3,MPI_COMM_WORLD,&status);
  MPI_Recv(&Rs,Lx*Ly*Lz,MPI_INT,my_id-1,4,MPI_COMM_WORLD,&status);
```

```
    MPI_Send(&energy,1,MPI_FLOAT,my_id-1,5,MPI_COMM_WORLD);
    MPI_Send(&r,1,MPI_FLOAT,my_id-1,6,MPI_COMM_WORLD);
    MPI_Send(&beta,1,MPI_FLOAT,my_id-1,7,MPI_COMM_WORLD);
    MPI_Send(&s,Lx*Ly*Lz,MPI_INT,my_id-1,8,MPI_COMM_WORLD);


}

    MPI_Barrier(MPI_COMM_WORLD);

//  printf("%d, %d, %f, %f \n", T, my_id, r, Rr);

    //sleep(10);
//  printf(" my_rank finished, %d \n", my_id);
    if(my_id%2 == 0) r = Rr;
    float PTswap = (Renergy-energy) * (Rbeta-beta);
    PTswap = expf(PTswap);

    if(PTswap >= r)
    {
        for(j=0;j<Lx*Ly*Lz;j++) s[j] = Rs[j];
    }

//printf("end 10 & !20 loop, %d, %d, \n", my_id, T);

}
```

```
  MPI_Barrier(MPI_COMM_WORLD);

if(T%10 == 0 && T%20 ==0)
{

float energy;
energy = Etot();
//float r = (float)rand()/(float)RAND_MAX;
float r = ran2(&seed);

float Renergy;
float Rr;
float Rbeta;
int   Rs[Lx*Ly*Lz];

//caution the first and last are excluded
//printf("in 10 & 20 loop, %d, \n", my_id);
if(my_id%2 != 0 && my_id != nprocs-1)
{
  MPI_Send(&energy,1,MPI_FLOAT,my_id+1,1,MPI_COMM_WORLD);
  MPI_Send(&r,1,MPI_FLOAT,my_id+1,2,MPI_COMM_WORLD);
  MPI_Send(&beta,1,MPI_FLOAT,my_id+1,3,MPI_COMM_WORLD);
  MPI_Send(&s,Lx*Ly*Lz,MPI_INT,my_id+1,4,MPI_COMM_WORLD);
```

```
    MPI_Recv(&Renergy,1,MPI_FLOAT,my_id+1,5,MPI_COMM_WORLD,&status);
    MPI_Recv(&Rr,1,MPI_FLOAT,my_id+1,6,MPI_COMM_WORLD,&status);
    MPI_Recv(&Rbeta,1,MPI_FLOAT,my_id+1,7,MPI_COMM_WORLD,&status);
    MPI_Recv(&Rs,Lx*Ly*Lz,MPI_INT,my_id+1,8,MPI_COMM_WORLD,&status);
  }
  else if(my_id%2 == 0 && my_id != 0)
  {
    MPI_Recv(&Renergy,1,MPI_FLOAT,my_id-1,1,MPI_COMM_WORLD,&status);
    MPI_Recv(&Rr,1,MPI_FLOAT,my_id-1,2,MPI_COMM_WORLD,&status);
    MPI_Recv(&Rbeta,1,MPI_FLOAT,my_id-1,3,MPI_COMM_WORLD,&status);
    MPI_Recv(&Rs,Lx*Ly*Lz,MPI_INT,my_id-1,4,MPI_COMM_WORLD,&status);



    MPI_Send(&energy,1,MPI_FLOAT,my_id-1,5,MPI_COMM_WORLD);
    MPI_Send(&r,1,MPI_FLOAT,my_id-1,6,MPI_COMM_WORLD);
    MPI_Send(&beta,1,MPI_FLOAT,my_id-1,7,MPI_COMM_WORLD);
    MPI_Send(&s,Lx*Ly*Lz,MPI_INT,my_id-1,8,MPI_COMM_WORLD);


  }


    MPI_Barrier(MPI_COMM_WORLD);
  //   printf(" my_rank finished, %d \n", my_id);
  if(my_id != 0 && my_id != nprocs-1)
  {
    if(my_id%2 == 0) r = Rr;
    float PTswap = (Renergy-energy) * (Rbeta-beta);
    PTswap = expf(PTswap);
```

```
  if(PTswap >= r)
  {
        for(j=0;j<Lx*Ly*Lz;j++) s[j] = Rs[j];
  }
}
//printf("end 10 & 20 loop, %d, %d, \n", my_id, T);



}



//end of PT swap





//every





//printf(" my_rank, T,  %6d, %6d, \n", my_id, T );
//

//PT swap
```

```
if(T > TotTime/2 && T%100 == 0)

{

float SiSj[NS*NS];

chi(SiSj);




for(int l = 0; l < NS*NS; l++) SiSjave[l] = SiSjave[l] + SiSj[l];





//printf("sisj, %f \n", SiSj[0]);



float m = mag();

//printf(" m %f \n", m );


mave = mave + m ;

mave2 = mave2 + m*m ;

mave4 = mave4 + m*m*m*m ;



float e = Etot();

//printf(" m %f \n", m );
```

```
eave = eave + e ;
eave2 = eave2 + e*e ;

//float chipp = chi();
//chippave = chippave + chipp;

counter = counter + 1;

}

}

mave = mave /counter;
mave2 = mave2 /counter;
mave4 = mave4 /counter;

eave = eave /counter;
eave2 = eave2 /counter;
```

```
chippave = chippave / counter;


float buffer[10];

buffer[0] = (float)ir;

buffer[1] = beta;

buffer[2] = mave;

buffer[3] = mave2;

buffer[4] = mave2-mave*mave;

buffer[5] = 1.0 - mave4/(3.0*mave2*mave2);

buffer[6] = eave;

buffer[7] = eave2;

buffer[8] = beta*(eave2-eave*eave);

chippave = 0.0;

for(int l = 0; l < NS*NS; l++) chippave = chippave + SiSjave[l] / counter;

buffer[9] = chippave;

for(int l = 0; l < NS*NS; l++) SiSjave[l] =  SiSjave[l] / counter;






printf(" %f, %f, %f, %f, %f, %f, %f, %f, %f, %f, \n",

buffer[0],buffer[1],buffer[2],buffer[3],buffer[4],buffer[5],buffer[6],buffer[7],buffer[8],buffer[9] );


/*

fprintf(f1, "%f %f \n", buffer[0], buffer[1]);

for(int i = 0; i < NS*NS; i++)

{
```

```
fprintf(f1, " %f \n",   SiSjave[i]);
}
*/


/*
for(int ii=0; ii < NS*NS; ii++)
 {
  int i1 = ii%NS;
  int i2 = ii/NS;
  int k1 = i1*NS + i2;
  int k2 = i2*NS + i1;
  printf("%d, %d, %d, %d, %d, %f, %f \n", T, my_id, ii, k1, k2, SiSjave[k1], SiSjave[k2]);

 }
getchar();
*/




//printf(" %5d, %6f, %f, %f, %f, %f, %f, %f, %f, %f \n", my_id, 1.0/beta, mave, mave2, mave2-mave*mave,
1.0 - mave4/(3.0*mave2*mave2), eave, eave2, beta*(eave2-eave*eave), chippave );
for(i=0;i<NS*NS;i++)
{
 SiSj2_Rave[i] = SiSj2_Rave[i] + SiSjave[i] * SiSjave[i];
 SiSj_Rave[i] = SiSj_Rave[i] + SiSjave[i] ;


}
```

```
MPI_Barrier(MPI_COMM_WORLD);

}


fprintf(f1, "%f  \n", beta);

for(int i = 0; i < NS*NS; i++)

{

fprintf(f1, " %f %f \n",   SiSj2_Rave[i]/Nr, SiSj_Rave[i]/Nr);

}


MPI_Finalize();
```

References

[1] Katzgraber, Helmut. "Introduction to Monte Carlo Methods." *Lecture at the Third International Summer School "Modern Computation Science"* (2011). *Arxiv*. Web. 24 Mar. 2015. <http://arxiv.org/abs/0905.1629>.

[2] Binder, K., and A.P Young. "Spin Glasses: Experimental Facts, Theoretical Concepts, and Open Questions." *Rev. Mod. Phys.* 58: 801. Web.

<http://journals.aps.org/rmp/pdf/10.1103/RevModPhys.58.801>