Exam 2012 Computational Physics

Morten Vassvik [*]

Norwegian University of Science and Technology, Trondheim

The second order phase transitions of the Q-valued Potts model for $Q = 2$ and $Q = 3$ has been investigated by having the spins in the Potts model placed on a square lattice in a temperature gradient. This way I have been able identify the ordered and disordered regions of the system by means of detecting *damage spread* across the lattice. I use the *heat bath algorithm* to evolve the system towards steady state. I have calculated the *correlation length critical exponent nu*, which diverges when the system undergoes a second order phase transition. I have also calculated the corresponding temperature where this transition occurs. $\nu = 1.018 \pm 0.024$ and $T_c = 1.1899 \pm 0.0079$ was found for the Q=2 system, while $\nu = 1.550 \pm 0.0274$ and $T_c = 1.04188 \pm 0.00194$ was found for the Q=3 system.

Introduction

For $Q \leq 4$, the Potts model undergoes a *second order phase transition* at a critical temperature $T = T_c$. For $Q \leq 4$, the phase transition is of second order and for $Q > 4$ the transition is of first order. In this paper I examine the cases where $Q = 2$ and $Q = 3$. A special characteristic of a second order phase transition is that the *correlation length* ξ diverges as:

$$\xi = a\,|T - T_c|^{-\nu} \tag{1}$$

where a is a prefactor and ν is the *correlation length critical exponent*. The correlation length in a first order phase transition, on the other hand, does not diverge, but spontaneously jumps to infinity.

Spins at a lower temperature than the *critical region* will be magnetized (ordered), while spins at higher temperature will be paramagnetized (disordered). The critical region is defined later. The critical region will be identified using the concept of *damage spreading*, which I use to measure the location and width of the critical region. I use these quantities to calculate T_c and ν.

Theory

Model

The Potts model is defined through the Hamiltonian

$$H = K \sum_{\langle i,j \rangle} \delta_{\sigma_i \sigma_j} \tag{2}$$

where the spins σ_i take the integer values $1 \leq \sigma_i \leq Q$, K is a positive constant, and $\delta_{\sigma_i,\sigma_j}$ is a Kronecker-Delta function. The sum is over nearest neighbours. For Q = 2, the Potts model is equivalent to the Ising model when $\sigma_i = \frac{1}{2}(3 + S_i)$, $J = \frac{K}{2}$ and $S_i = \pm 1$.

The spin system is on a 2D square lattice of size L. I have oriented the lattice so that the spin in the lower-left corner is at position $(0, 0)$, and the spin in the upper-right corner is at position $(L-1, L-1)$.

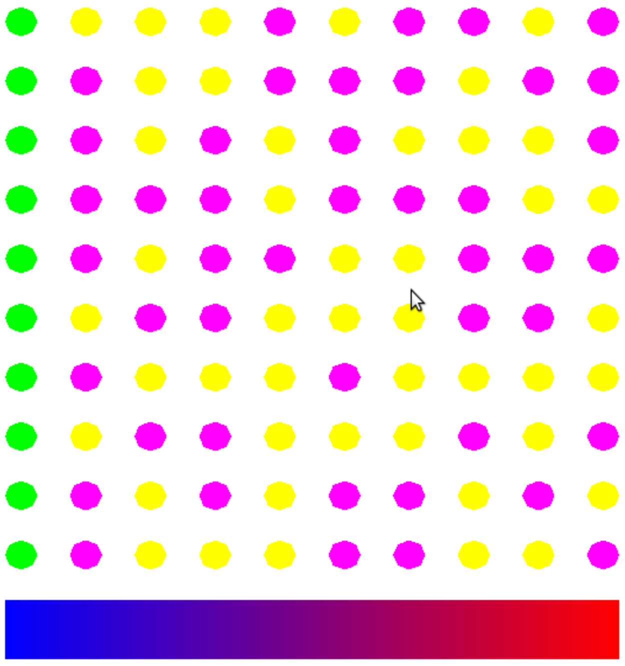


Figure 1: Two spin systems of size L = 8 superimposed on top of each other. Green means the spins are different. Purple means spin up. Yellow means spin down. The color gradient at the bottom denotes the temperature gradient, from hot (red) to cold (blue).

The system is kept in a temperature gradient, where the temperatures along the first column and last column are kept constant at $T = T_{min}$ and $T = T_{max}$, respectively. The temperature varies linearly along the rows. The gradient is defined as:

$$g = \frac{T_{max} - T_{min}}{L - 1} \tag{3}$$

The temperature of each node (spin) is then defined as follows:

$$T_{(i,j)} = T_{min} + g \cdot i \tag{4}$$

where the index i goes along the rows and j along the columns.

[*]I have collaborated with Isak Buhl-Mortensen, Christian Roca Catalá and Amund Hov. I have discussed results with Vegard Flovik, Lalit Kumar and Eirik Lhaugen Fjrbu

A periodic boundary condition is implemented in the direction *perpendicular to* the temperature gradient, in the j-direction.

The concept of damage spreading works as follows: I prepare two systems, identical apart from the spins in the first column, with constant $\sigma^{(1)}_{(1,j)} = 1$ in the first system and constant $\sigma^{(2)}_{(1,j)} = 2$ in the second. The same spins are updated in both systems during the Monte Carlo updating scheme. The first system represents a system with a small magnetic field at the first column and clusters of spin 1 tend to form close to it. Likewise for the second system. The spins at the last column are also held constant, but this is not strictly necessary.

Define a quantity $d_{(i,j)}$ which is 1 if the spins in the two systems are different and 0 if they are identical. I use this to identify the interface that seperates the ordered from the unordered region: For each row j, search for the first $d_{(i,j)}$ with value 1 starting from the $i = L-1$ edge. This is the *profile* of the interface. The profile is characterized by its mean position (along the row) and its width (fluctuation):

$$\langle I \rangle = \frac{1}{L}\sum_{j=0}^{L-1} I(j) \tag{5}$$

$$W^2 = \frac{1}{L}\sum_{j=0}^{L-1} \left(I(j) - \langle I \rangle\right)^2 \tag{6}$$

where each I(j) is measured at a given Monte Carlo "time" (sample time, *not* physical time).

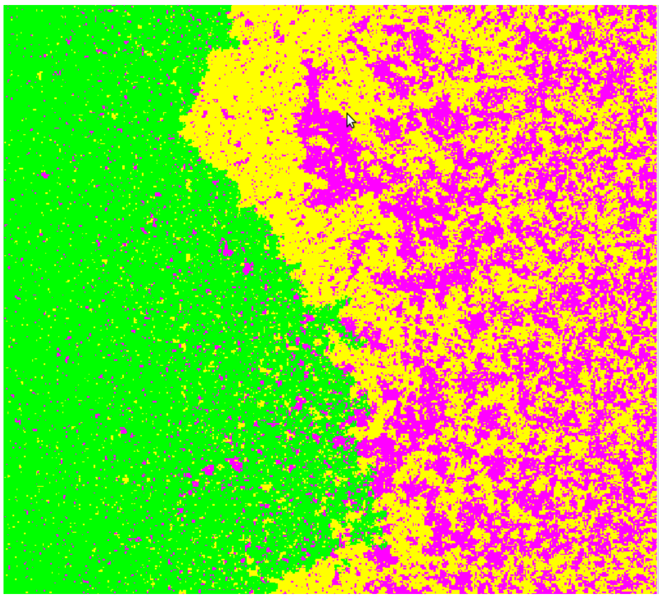


Figure 2: An interface profile of a large system at a given MC sampling time given by the border of the green cluster.

I use the theory by Sapoval, Rosso, and Gouyet to determine the correlation length exponent ν. Since the interface that separate the ordered from the disordered region fluctuates across the critical region, I have that $W = \xi$, the correlation length. The temperature difference across this region is given by

$$\Delta T = gW \tag{7}$$

Thus, by combining equation (1) and (7):

$$W = a\left|\Delta T\right|^{-\nu} = a\left|gW\right|^{-\nu} \tag{8}$$

which I can solve for W and get:

$$W = bg^{-\frac{\nu}{1+\nu}} \tag{9}$$

where $b = a^{1/(1+\nu)}$. Therefore, to find ν I plot W as a function of the gradient g in a log-log plot, whose slope I use to extract the critical exponent.

Due to the finite size of the system, I cannot determine the critical temperature directly. I determine an effective critical temperature, T_{eff}, which I can define by combining equation (4) and (6):

$$T_{eff} = T_{min} + g\langle I \rangle \tag{10}$$

I extrapolate the lattice to infinity to find T_c. By using *finite size scaling analysis*, I find:

$$T_{eff} = T_c + Ag^{\frac{1}{1+\nu}} \tag{11}$$

Thus, if I plot T_{eff} as a function of $g^{1/(1+\nu)}$, I can find T_c by setting $g = 0$ ($L = \infty$)

Errors and Limitations

Due to the limited size of the systems, limitated simulation time and the fact evolution of the system strongly depend on the initial configuration, I am bound to get some errors due to fluctuations. I will try to discuss some of them here.

The size of the system scales as L^2, that means that the probability to choose an arbitrary spin at random is $\frac{1}{L^2}$. This also means that if I double the system size, I most likely need to do at least four times as many Monte Carlo steps to start the measurements. Most likely I need to do more, because of other effects and correlations. I need to run the system until it has reached a *steady state*, that is, the average position of the interface, and therefore the fluctuations W and the effective temperature T_{eff}, will stay approximitely constant over time. The time until reaching this steady state, t_SS, varied quite a lot. For systems of size $L = 100$, I had to wait between *50 million* and *200 million* Monte Carlo steps until I could start measuring. As the systems grow even larger, this *time* increases dramatically: For instance, I encountered a system of size L=500 that did not reach steady state even after *15 billion* steps. Therefore, to do simulations on large systems requires a lot of time and resources. It is essential to identify when the system is in, or close to, steady state so that I know when I can start measuring.

From equation (9), I can see that the fluctuations of the interface increase as the gradient decreases (since ν is positive), this is a natural consequence of increasing the size of the system: If I keep the temperature constant, but double the size, ΔT must stay constant and naturally W must also increase. If I keep the size constant, but increase the temperature range, I can lower the fluctuations.

An observation worth mentioning is that the *fluctuation* in $\langle I \rangle$, W, and T_{eff} themselves are quite big, as can be seen from Figure 3. The significance of these lessen as the size of the system increases (and $W/\langle I\rangle \to 0$ as $g \to 0$). The constant spins at the hottest column work has the opposite effect as the spins at the coldest column. The effect of these lessen as the system size increase, but they need to be taken into account, since they are different for each initial configuration.

One last source of error may be the random number generator used to choose spin, specially if the period is too small. An example of this can be seen in Figure 3, where the low period of the RNG appear as a large-scale periodic fluctuation. In this example I used the infamous 16807-LCG to generate all my random numbers. I therefore decided to change from 16807 to the better suited Mersenne Twister algorithm[2] using the GNU Scientific Library, which has a period of 2^{19937} (compared to 2^{30} for 16807).

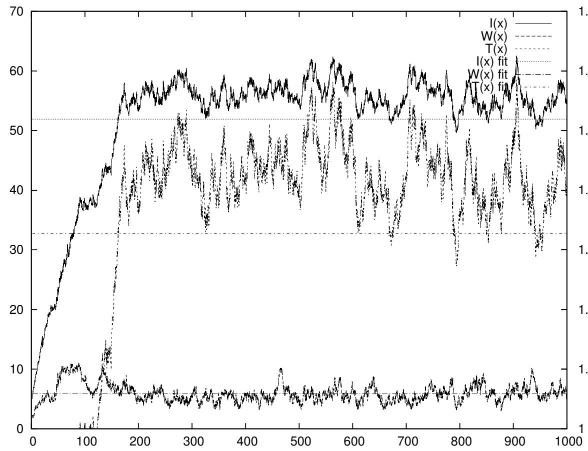


Figure 3: "Time" evolution of a system of size 100x100 with $T_{max} - T_{min} = 1.2$ run for 100 million Monte Carlo steps. The upper and lower graphs are the interface $\langle I\rangle$ and the fluctuations W, respectively, both using the left y-axis. The middle graph is T_{eff}, calculated from equation (10), and uses the right y-axis. The constant fits are off because I started sampling from the first iterations. The x-axis has units (10 000x) Monte Carlo steps. Steady state was reached at about 20 million iterations.

Implementation

For these simulations, I work in units where the constant in the Hamiltonian, K, is 1. I use the *heat bath algorithm* as the Monte Carlo updating scheme, and it works as follows: First you choose a spin at random (sans the first and last colums) and give it a new value q, chosen so that $1 \le q \le Q$ and with probability proportional to the local Boltzmann weigth

$$\bar{p}(q) = e^{(\delta_{(i-1,j),q} + \delta_{(i,j-1),q} + \delta_{(i+1,j),q} + \delta_{(i,j+1),q})/T_{(i,j)}} \tag{12}$$

To do this, I form the sum

$$N = \sum_{q'=1}^{Q} \bar{p}(q') \tag{13}$$

to normalize the Boltzmann weights. We then calculate the cummulative probabilities

$$P(q) = \frac{1}{N} \sum_{q'=q}^{Q} \bar{p}(q') \tag{14}$$

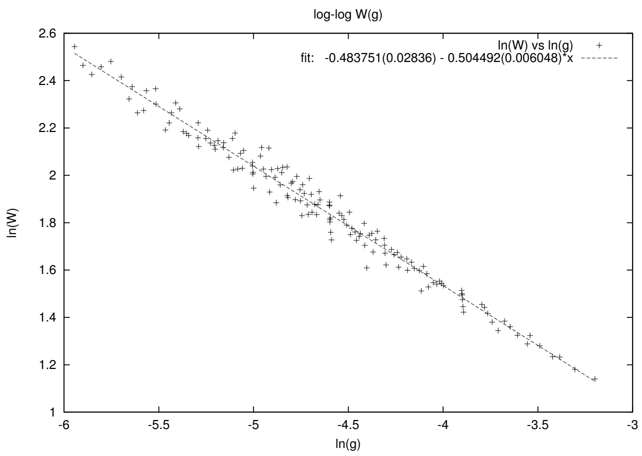
for $1 \le q \le Q$. We then pick a random number r uniformly distributed on the unit interval. If $r < P(1)$, set q = 1, otherwise if $P(n-1) \le r < P(n)$ set q = n. We note that the old spin value is not present in choosing the new value, only the adjacent spins are used[1].

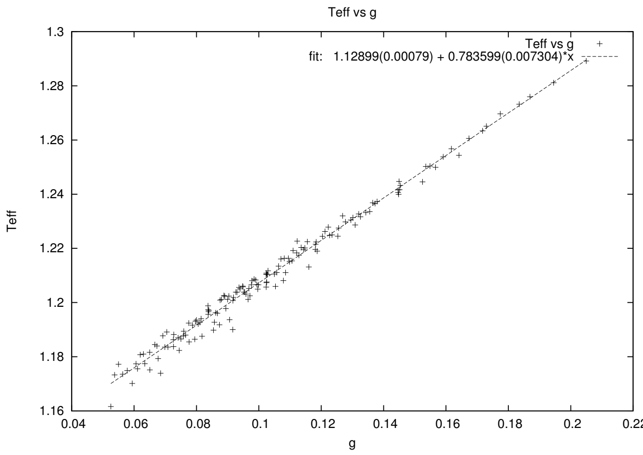
I let the system run until it has reached a steady state, that is, $\langle I\rangle$has reached a plateu and is fluctuating about its average value, which varies depending on L and the temperature interval. From this moment on, I calculate $\langle I\rangle$, W^2, and T_{eff} from equation (5),(6), and (10) respectively. I do 100 000 iterations of the heat bath algorithm between each sample, this is to get sufficient info about large-scale fluctiations. When I have gathered 5000 samples, I calculate the average value and standard deviation of $\langle I\rangle$, W^2, and T_{eff} from these, which I use with equation (9) and (11) in order to find ν and T_c. I vary the size of the system and the temperature range to get different values for the gradient g.

I used GNUPLOT's fit-function to fit my data to a linear function, where I provided the uncertainties in $\langle I\rangle$, W^2 and T_{eff} as a third argument in addition.

Results and Discussion

Q=2





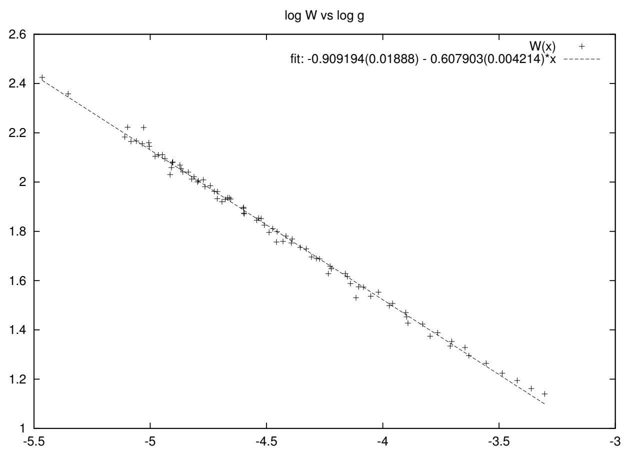
[1]This may be the origin of the name of the algorithm. The spins are located inside a "heat bath", surrounded by other spins of other temperatures.

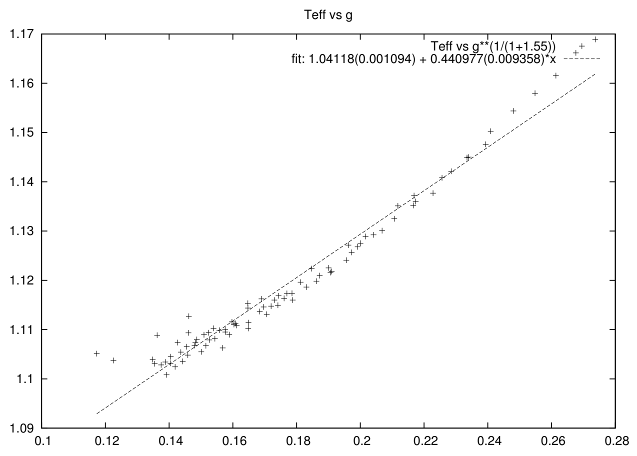
Figures 4 and 5: The log-log plot of equation (9) and the plot of equation (11), respectively, for Q=2. Regression with gnuplot gave $\nu = 1.018 \pm 0.024$ and $T_c = 1.12899 \pm 0.00079$

For Q = 2 I varied the system size from $L = 50$ to $L = 230$. At the same time, I varied the temperature range from $T_{max} - T_{min} = 0.6$ to 2.0 centered about $T_c = 1/ln(1+\sqrt{2})$. The log-log plot of equation (9) is showed in Figure 4, from which I calculate $\nu = 1.018 \pm 0.024$, which is within the theoretically expected value, $\nu = 1$.

In Figure 5 you can see the plot of T_{eff} vs the gradient ($\nu = 1$) with the same data points as was used to calculate ν above, using the value of ν found above. We find that $T_c = 1.12899 \pm 0.00079$, which is close to the expected theoretical value, $T_c = 1/\ln(1+\sqrt{2}) \approx 1.134$.

Q=3





Figures 6 and 7: The log-log plot of equation (9) and the plot of equation (11), respectively, for Q=3. Regression with gnuplot gave $\nu = 1.550 \pm 0.0274$ and $T_c = 1.04188 \pm 0.00194$

For Q = 3 I varied the system size from $L = 50$ to $L = 190$. At the same time, I varied the temperature range from $T_{max} - T_{min} = 0.6$ to 1.8 centered about $T_c = 1/\ln(1+\sqrt{3}) \approx 0.99$. The log-log plot of equation(9) is showed in Figure 6, from which I calculate $\nu = 1.550 \pm 0.0274$, which is within the theoretically expected value, $\nu = 1$.

In Figure 7 you can see the plot of T_{eff} vs the gradient ($\nu = 1$) with the same data points as was used to calculate ν above using the value of ν found above. We find that $T_c = 1.04188 \pm 0.00194$, but I do not have a theoretical value to compare it with. I have my doubts that this is completely correct, since the points does deviate quite a bit from a straight line. This could mean that my value of ν is slighty off. I unfortunately do not have enough data to make any more statements about this. If I had more time, I would definitively investigate this further. Perhaps the *burning algorithm* explained in [1] would give different results.

Other observations

The time until the system reach steady state varies greatly, and the variation increases dramatically with increased system size. For L = 100, I have observed values of t_{SS} ranging from 20 million iterations until 200 million iterations. For L = 200, I observed values from 100 million to 600 million. For L = 500, I had one case where it did not reach steady state, even after 15 billion iterations. This seems to depend strongly on the initial configuration of the spins, especially after the first clusters have formed in the ordered region. When the interface front is very close to T_{min}, it is going to compete with large clusters of identical spins. The larger these clusters are, the longer it seems to take for it to reach steady state.

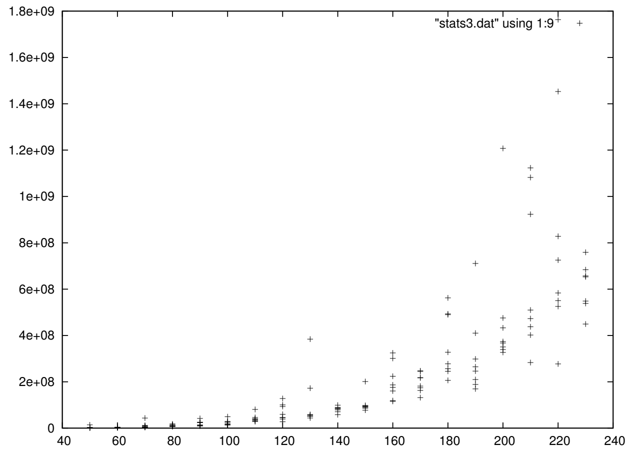


Figure 8: Time until steady state as a function of system size.

In Figure 8 you can see a plot of t_{SS} vs system size for Q = 2, where the criterion $T_{eff} > T_c$ was used as a criterion for when to start sampling. When I calculated ν and T_c, I used the following criterion to determine when to start sampling: I tried a few trial runs to observe how much t_{SS} fluctuated for a given size L. I then used these numbers to set a roof of when to start sampling, varying between 50 million iterations for small L (< 75), and up to 800 million iterations for large L (> 200). I realize that this is not an efficient way to decide if the system is ready to be sampled, since I may risk waiting longer than necessary before starting.

I also measured the fluctuation of the average values of $\langle I \rangle$, W, and $Teff$. The following were observed: The flucttutations in $\langle I \rangle$ and W went down for increasing temperature intervals. The opposite happened for T_{eff}, where its standard deviation increased with increasing temperature interval. Both of these make sense in the light of equation (10) and (11). In Figure 9, where T_{eff} is plotted as a function of the gradient with error bars for the y-values, you can see how severe the fluctuations are. This means that the large uncertainty in the averages themselves should be taken into account. I have regrettably not been able to do this, due to the lack of time. For further research I would very much prioritize doing the sampling process more properly and do the statistics correctly, and most certainly do the regression correctly while correctly taking into account all the different kind of averages and uncertainties.

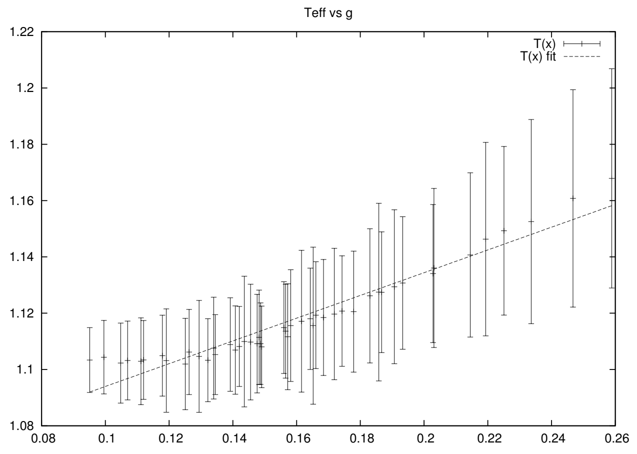


Figure 9: T_{eff} for Q = 3 with errorbars.

I have put several videos and plots on my home page[2]. In the videos you can see the "time evolution" of the system with plots attached, from the initial configuration and some time after steady state has been reached.

Conclusion

I have shown that the Q-states Potts model can be simulated and sampled by using heat bath algorithm. It seems to give good results for Q = 2, and it also seems to work to a certain extent for Q = 3. More research and data gathering must be done to determine the reason for the deviations from equation (11) for Q=3.

References

[1] A. Hansen and D. Stauffer, The three-dimensional Ising model in a temperature gradient, Physica A, 189,611(1992)

[2] M. Matsumoto and T. Nishimura Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998)

[2]folk.ntnu.no/mortevas/

```
#include <iostream>
#include <cmath>
#include <ctime>
#include <fstream>
#include <gsl/gsl_rng.h>

using namespace std;

const int Q = 3;
const double PI = 4*atan(1.0);
const double Tc = 1 / log(1 + sqrt(Q));
gsl_rng *r = gsl_rng_alloc (gsl_rng_mt19937);   //period = 2^19937 - 1

bool FIRST;
int sampInt,samps,firstSamp,L;
long long iter;
double dT, g, Tmax, Tmin, Iavg, Wavg, Istd, Wstd, Tavg, Tstd;

int * spin1, * spin2;
double * temp;
int ** n;
double * iFace,* wFace,* tFace;

int randIBM();
bool seedIBM(unsigned int seed);
int randIBM();
double randIBMf();
short randIBMs(int Q);

void heatBath();
void calcQuantities(ofstream &file1, int & samp);

int main(int iArgc, char** cppArgv){
    ofstream f2("stats.dat");
    sampInt = 1e5;
    samps = 5000;
    iFace = new double [samps];
    wFace = new double [samps];
    tFace = new double [samps];

    for (L = 25; L <= 125; L += 10 ){
        cout << "L␣=␣" << L << endl;
        spin1 = new int[L*L];
        spin2 = new int[L*L];
        temp = new double[L];
        n = new int * [L*L];
        for (int ij = 0; ij < L*L; ij++){
            n[ij] = new int[5];
        }
        for (dT = 0.2; dT <= 1.4; dT += 0.1){
            cout << "dT␣=␣" << dT << "␣" << flush;
            iter = 0;
            g = dT/(L-1);
            Iavg = 0.0,Wavg = 0.0,Istd = 0.0;
            Wstd = 0.0,Tavg = 0.0,Tstd = 0.0;
            FIRST = true;
            seedIBM(abs(randIBM()));

            for (int i = 0; i < L; i++){
                temp[i] = (Tc - dT/2) + g*i;
            }
            for (int i = 0; i < L; i++){
                for (int j = 0; j < L; j++){
                    int ij = j*L + i;
                    n[ij][0] = ij;
                    n[ij][1] = ((i+1) == L ? ij-(L-1) : ij+1);
                    n[ij][2] = ((j+1) == L ? ij-(L-1)*L : ij+L);
                    n[ij][3] = (i == 0 ? ij+(L-1) : ij-1);
                    n[ij][4] = (j == 0 ? ij+(L-1)*L : ij-L);

                    if (i != 0){
                        spin1[ij] = randIBMs(Q);
                        spin2[ij] = spin1[ij];
                    }
                    else{
                        spin1[ij] = 1;
                        spin2[ij] = 2;
                    }
                }
            }

            int i = 0;
            while (i <= samps){
                heatBath();
                calcQuantities(f2, i);
                if (i % 1000 == 0 && i > 0)
                    cout << i << "␣" << flush;
            }
            for (int i = 0; i < samps; i++){
                Iavg += iFace[i];
                Wavg += wFace[i];
                Tavg += tFace[i];
            }
            Iavg/=(samps);
            Wavg/=(samps);
            Tavg/=(samps);
            for (int i = 0; i < samps; i++){
                Istd += (iFace[i] - Iavg)*(iFace[i] - Iavg);
                Wstd += (wFace[i] - Wavg)*(wFace[i] - Wavg);
                Tstd += (tFace[i] - Tavg)*(tFace[i] - Tavg);
            }
            Istd= sqrt(Istd/samps);
            Wstd= sqrt(Wstd/samps);
            Tstd= sqrt(Tstd/samps);

            f2 << L << "␣" << dT << "␣" << Iavg << "␣" << Istd << "␣"
                 << Wavg << "␣" << Wstd << "␣"
                 << Tavg << "␣" << Tstd << "␣"
                 << firstSamp << "␣" << g << endl;
            cout << "done" << endl << endl;

        }
        delete []spin1;
        delete []spin2;
        delete []temp;
        for (int i = 0; i < L*L; i++){
            delete []n[i];
        }
        delete []n;
    }
    delete []iFace;
    delete []wFace;
    delete []tFace;
    f2.close();

    return 0;
}

int randIBM(){
    static int IBM = time(0)*2 + 1;
    return (IBM*16807);
}
bool seedIBM(unsigned int seed){
    gsl_rng_set(r,seed);
}
double randIBMf(){
    return (gsl_rng_uniform (r));
}
short randIBMs(int Q){
    return (1 +  int(Q*randIBMf() ));
}

void heatBath(){
    int x, y, q, q1, q2, ij;
    double r, dE, N1, N2;
    double * pbar, * p, * P, *pbar2, *p2, *P2;

    pbar = new double[Q];
    p = new double[Q];
    P = new double[Q];
    pbar2 = new double[Q];
    p2 = new double[Q];
    P2 = new double[Q];

    for (int samp = 0; samp < sampInt; samp++){

        x = int((L - 2)*randIBMf() + 1);
        y = int(L*randIBMf());
        ij = y*L + x;
        q = randIBMs(Q);
        r = randIBMf();
        N1 = 0;
        N2 = 0;
        for (int k = 0; k < Q; k++ ){
            dE = 0;
            dE += (spin1[n[ij][1]] == k+1 );
            dE += (spin1[n[ij][2]] == k+1 );
            dE += (spin1[n[ij][3]] == k+1 );
            dE += (spin1[n[ij][4]] == k+1 );
            pbar[k] = exp(dE/temp[x]);
            N1 += pbar[k];
            dE = 0;
            dE += (spin2[n[ij][1]] == k+1 );
            dE += (spin2[n[ij][2]] == k+1 );
            dE += (spin2[n[ij][3]] == k+1 );
            dE += (spin2[n[ij][4]] == k+1 );
            pbar2[k] = exp(dE/temp[x]);
            N2 += pbar2[k];
        }
        for (int k = 0; k < Q; k++){
            p[k] = pbar[k]/N1;
            p2[k] = pbar2[k]/N2;
        }
        for (int k = 0; k < Q; k++){
            P[k] = 0.0;
            P2[k] = 0.0;
            for (int l = 0; l <= k; l++){
                P[k] += p[l];
                P2[k] += p2[l];
            }
        }
        for (int k = 0; k < Q; k++){
            if (r < P[k]){
                q1 = k+1;
                break;
            }
        }
        for (int k = 0; k < Q; k++){
            if (r < P2[k]){
                q2 = k+1;
                break;
            }
        }
        spin1[ij] = q1;
        spin2[ij] = q2;
        iter++;
    }
    delete [] pbar;
    delete [] p;
    delete [] P;
    delete [] pbar2;
    delete [] p2;
    delete [] P2;
}

void calcQuantities(std::ofstream &file1, int& samp){
    static int MAX;
    if (L < 100)
        MAX = 2e8;
    else if (L < 150)
        MAX = 4e8;
    else if (L < 200)
        MAX = 5e8;
    else if (L < 250)
        MAX = 6e8;
    else
        MAX = 8e8;
    double I = 0, W2 = 0;
    for (int j = 0; j < L; j++ ){
        int i=0;
        for (i = L-1; i > 0;i--){
            if (spin1[j*L + i] != spin2[j*L + i])
                break;
        }
        I += (i+1);
    }
    I /= L;
    if (iter > MAX && temp[0] + g*(I) > Tc){
        for (int j = 0; j < L; j++){
            int i=0;
            for (i = L-1; i > 0;i--){
                if (spin1[j*L + i] != spin2[j*L + i])
                    break;
            }
            W2+= (i+1 - I)*(i+1 - I);
        }
        W2 /= L;
        if (FIRST){
            FIRST = false;
            firstSamp = iter;
         }
        iFace[samp] = I;
        wFace[samp] = sqrt(W2);
        tFace[samp] = temp[0] + g*(I);
        samp++;
    }
}
```