Monte Carlo Simulation of Potts Model: Assignment 3

Kevin Grainger

December 16, 2024

Contents

1 Abstract

The Potts model is a generalization of the Ising model, describing spin configurations on a two-dimensional lattice. In this report, we employ the Metropolis algorithm to simulate the Potts model for $q = 3$ and $q = 5$, estimate magnetization ($\langle M \rangle$), and ($\langle M^2 \rangle$). We will then analyze the phase transitions at different values of β (inverse temperature).

2 Background and Theory

2.1 The Potts Model

The Potts model represents the spins within a lattice. Each site on a 2D grid takes one of q possible spin values.

$$\sigma_{x,y} \in \{1, 2, \dots, q\}.$$

The Hamiltonian for such a system would be as follows:

$$H = -J \sum_{i,j} \delta_{\sigma_i, \sigma_j}$$

Instead of using this Hamiltonian our approach is to count the number of 'frustrated bonds', this refers to the number of mismatched bond in our system. The Potts model is now characterized by the Action:

$$S(\sigma) = \frac{1}{2} \sum_{x,y} \sum_{(x',y') \in n(x,y)} (1 - \delta_{\sigma_{x,y}, \sigma_{x',y'}}),$$

where $n(x, y)$ represents the four nearest neighbors of the site (x, y), and δ is the Kronecker delta.

2.2 Thermodynamics of the Potts Model

2.2.1 Magnetization and Observables

The magnetization is the thermodynamic quantity we will use to illuminate the state of our system. Magnetization measures the alignment of the spins, and is thus related directly to the Action of the system. We will need to estimate $< M >$ and $< M^2 >$, and thus the variance.
The magnetization of a configuration is defined as:

$$M(\sigma) = \frac{q f(\sigma) - 1}{q - 1},$$

where $f(\sigma)$ is the fraction of lattice sites in the most frequent spin state:

$$f(\sigma) = \max_{k \in \{1, \dots, q\}} \frac{1}{L^2} \sum_{x,y} \delta_{k, \sigma_{x,y}}.$$

With knowledge of magnetization and its' variance, we can see where the system undergoes a phase transition.

2.2.2 Phase Transitions

A phase transition is the change in state of a matter system as we change some variable. In our case we changing β, and thus changing temperature as a result. There are some expected behaviors we should see in our simulations, these are as follows.
For q=2, the Ising model, we have the phases:

1. The Ordered Phase- high β - aligned spins - high magnetization
2. The Disordered Phase - low β - random spins - low magnetization

The phases in the Potts model, with $q \leq 4$, are continuous, so our variables will change smoothly. For $q > 4$ we have a discontinuous transition, our variables will change abruptly.
For $q = 3$, the Potts model undergoes a second-order phase transition, while for $q = 5$, it exhibits a first-order phase transition. These transitions are reflected in the behavior of $\langle M \rangle$ as a function of β.

2.3 Monte Carlo Methods and Markov Chain

A Monte Carlo Method is a label given to large group of functions which use repeated random events to estimate the possible outcomes of a future event. By using a given probability distribution, the Monte Carlo method can predict the future values of a variable by making random alterations to the system. The process can be generalized as such:

1. Propose random change
2. Accept of deny change
3. Measure as the system progresses.

The Markov Chain is the logical framework of our Potts Model simulation. The Markov chain is sequence of states, in which the probability oof transitioning to the next state depends only on the previous states. The aim is to create a random and memory less chain of events.
In our chase, when we wish to transition from one spin state to the next, we first need to propose the change in spin, we then need to create criteria in which to decide wether or not to make the transition. To do this we consider that the system naturally favors the lower energy states, thus any random move that increases Action (energy) will have a reduced probability of occurring. The exact relationship is $\propto exp(-\beta \Delta S)$ Our procedure is then as follows:

1. Propose a transition (randomly select a spin to change)
2. Compute the change in action this change would make (compute change in Energy)
3. Accept the move if $\Delta S \leq 0$, otherwise make transition with probability $exp(-\beta \Delta S)$
4. For calculating observable, disregard the first steps and allow our system to diverge properly from the starting conditions, then we can make more accurate measurements.

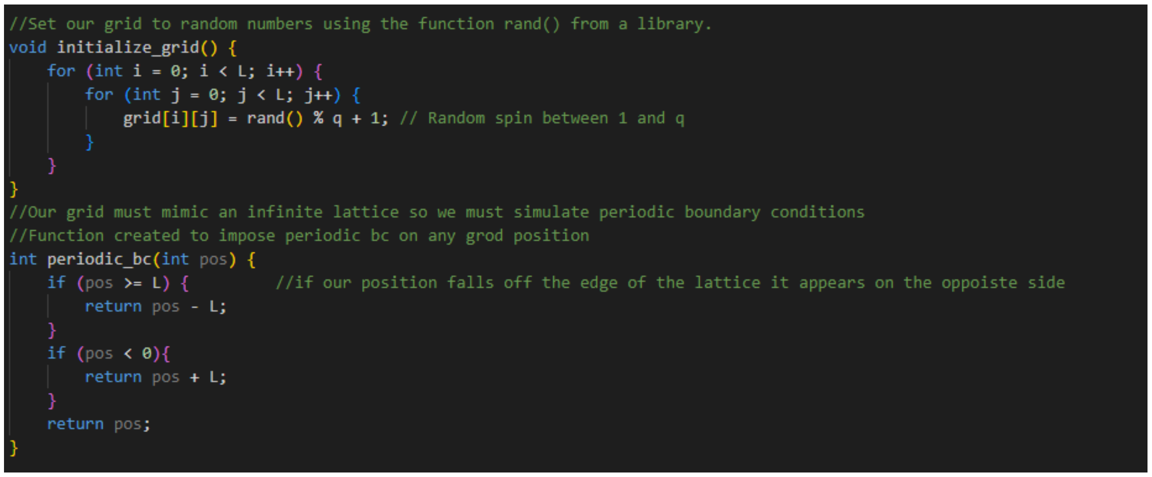
3 Methods

3.1 Simulation Setup

3.1.1 Lattice and Boundary conditions

The boundary conditions in our lattice must be periodic, $\sigma_{x+L} = \sigma_x$, this is achieved using the modulo or by adding/subtracting a lattice length L. In the case a co-ordinate is not within our lattice range, we can let it 'wrap' around and appear in our lattice from the opposite end, the position in the lattice is thus dependent on the remainder after division by Lattice length L.

- Lattice size: 24×24.
- Number of steps: $10,000$.
- Thermalization: First $4,000$ to let out system 'settle' and distance itself from the initial conditions, this value was found through trial and error.
- β range: $[0.5, 1.5]$ with 50 values.
- Value of q =3 or 5 . This needs to be inputted.

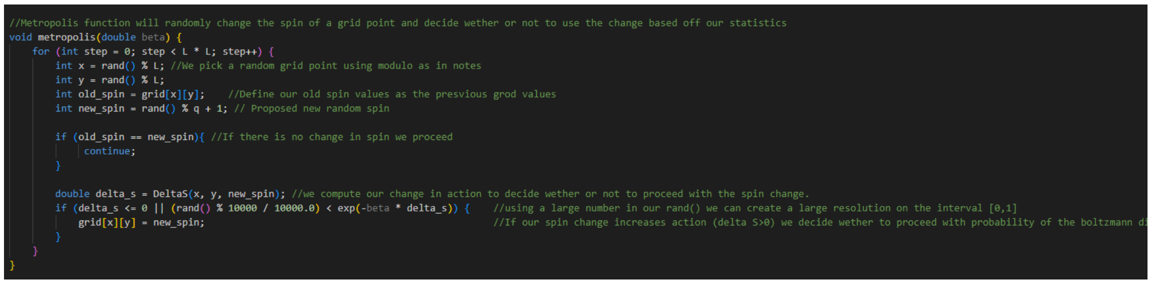


```
//Set our grid to random numbers using the function rand() from a library.
void initialize_grid() {
    for (int i = 0; i < L; i++) {
        for (int j = 0; j < L; j++) {
            grid[i][j] = rand() % q + 1; // Random spin between 1 and q
        }
    }
}
//Our grid must mimic an infinite lattice so we must simulate periodic boundary conditions
//Function created to impose periodic bc on any grod position
int periodic_bc(int pos) {
    if (pos >= L) {         //if our position falls off the edge of the lattice it appears on the oppoiste side
        return pos - L;
    }
    if (pos < 0){
        return pos + L;
    }
    return pos;
}
```

Figure 1: Initialization of our system

3.1.2 Metropolis Algorithm

The Metropolis Algorithm is a Monte Carlo method, which uses a Markov chain. New states are added to the system based only on the previous state. First the new state is 'proposed', it can then be accepted of rejected based off the value of the probability distribution at that point. If all is done correctly we should have a sequence of events from which the probability distribution could be derived from. So our Potts simulation should 'reveal' the Boltzmann distribution.

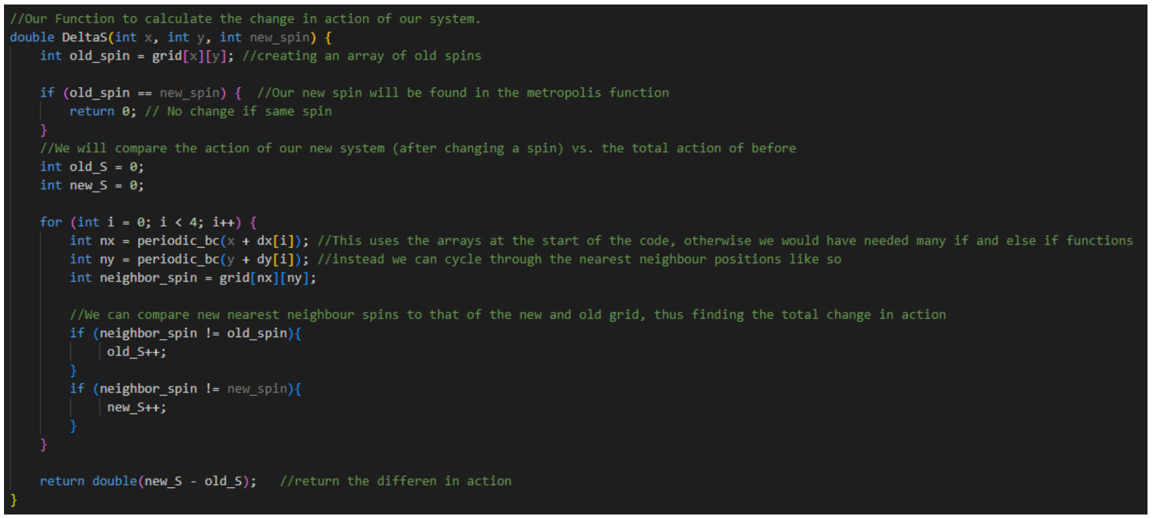


```
//Metropolis function will randomly change the spin of a grid point and decide wether or not to use the change based off our statistics
void metropolis(double beta) {
    for (int step = 0; step < L * L; step++) {
        int x = rand() % L; //We pick a random grid point using modulo as in notes
        int y = rand() % L;
        int old_spin = grid[x][y];    //Define our old spin values as the presvious grod values
        int new_spin = rand() % q + 1; // Proposed new random spin

        if (old_spin == new_spin){ //If there is no change in spin we proceed
            continue;
        }

        double delta_s = DeltaS(x, y, new_spin); //we compute our change in action to decide wether or not to proceed with the spin change.
        if (delta_s <= 0 || (rand() % 10000 / 10000.0) < exp(-beta * delta_s)) {    //using a large number in our rand() we can create a large resolution on the interval [0,1]
            grid[x][y] = new_spin;                                                  //If our spin change increases action (delta S>0) we decide wether to proceed with probability of the boltzmann di
        }
    }
}
```

Figure 2: Metropolis function, makes a random spin change and decides the step forward

3.2 Total Action of our system

By comparing the affect of a spin change on the neighboring grid points, we can calculate the total change in action of our system and thus the change in energy. The code used here is highly condensed, using an array dx[] and dy[], storing the offset of each nearest neighbor form the grid point of interest.The functions DeltaS and Metropolis are linked, as the values for the new spins are required for this function.

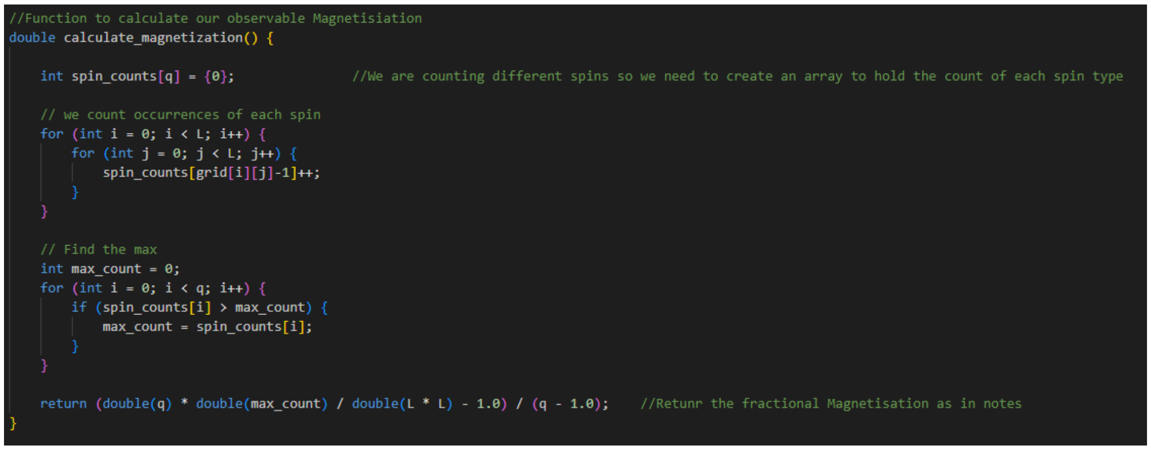


```
//Our Function to calculate the change in action of our system.
double DeltaS(int x, int y, int new_spin) {
    int old_spin = grid[x][y]; //creating an array of old spins

    if (old_spin == new_spin) {  //Our new spin will be found in the metropolis function
        return 0; // No change if same spin
    }
    //We will compare the action of our new system (after changing a spin) vs. the total action of before
    int old_S = 0;
    int new_S = 0;

    for (int i = 0; i < 4; i++) {
        int nx = periodic_bc(x + dx[i]); //This uses the arrays at the start of the code, otherwise we would have needed many if and else if functions
        int ny = periodic_bc(y + dy[i]); //instead we can cycle through the nearest neighbour positions like so
        int neighbor_spin = grid[nx][ny];

        //We can compare new nearest neighbour spins to that of the new and old grid, thus finding the total change in action
        if (neighbor_spin != old_spin){
            old_S++;
        }
        if (neighbor_spin != new_spin){
            new_S++;
        }
    }

    return double(new_S - old_S);   //return the differen in action
}
```

Figure 3: Computes the total action due to a change in random spin

3.3 Calculating fractional Magnetization

Given we needed to use 2 values

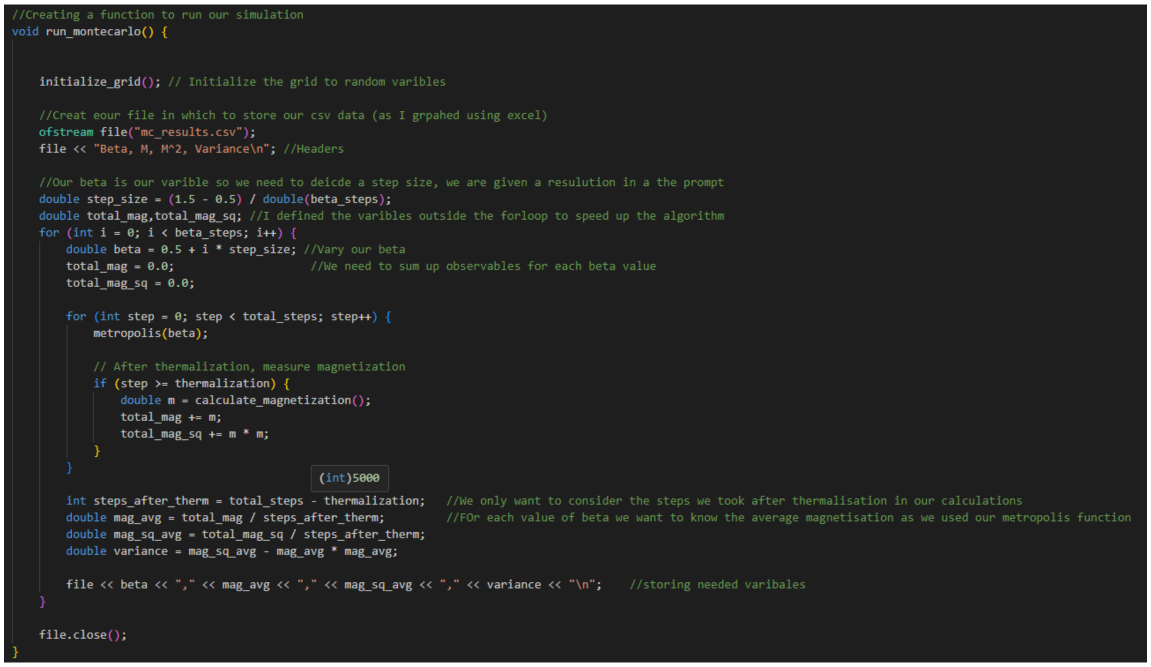


```cpp
//Function to calculate our observable Magnetisiation
double calculate_magnetization() {

    int spin_counts[q] = {0};               //We are counting different spins so we need to create an array to hold the count of each spin type

    // we count occurrences of each spin
    for (int i = 0; i < L; i++) {
        for (int j = 0; j < L; j++) {
            spin_counts[grid[i][j]-1]++;
        }
    }

    // Find the max
    int max_count = 0;
    for (int i = 0; i < q; i++) {
        if (spin_counts[i] > max_count) {
            max_count = spin_counts[i];
        }
    }

    return (double(q) * double(max_count) / double(L * L) - 1.0) / (q - 1.0);    //Retunr the fractional Magnetisation as in notes
}
```

Figure 4: Function to calculate the most frequent spin value and use this to return fractional Magnetization

3.4 Final Monte Carlo function

In this final function we call our metropolis function for different values of beta (thus varying temperature). We can thus compute values of $< M >$ and $< M^2 >$ for each values of beta. I did this by adding the magnetization for each beta value and averaging this value over the number of steps taken.



```cpp
//Creating a function to run our simulation
void run_montecarlo() {


    initialize_grid(); // Initialize the grid to random varibles

    //Creat eour file in which to store our csv data (as I grpahed using excel)
    ofstream file("mc_results.csv");
    file << "Beta, M, M^2, Variance\n"; //Headers

    //Our beta is our varible so we need to deicde a step size, we are given a resulution in a the prompt
    double step_size = (1.5 - 0.5) / double(beta_steps);
    double total_mag,total_mag_sq; //I defined the varibles outside the forloop to speed up the algorithm
    for (int i = 0; i < beta_steps; i++) {
        double beta = 0.5 + i * step_size; //Vary our beta
        total_mag = 0.0;                    //We need to sum up observables for each beta value
        total_mag_sq = 0.0;

        for (int step = 0; step < total_steps; step++) {
            metropolis(beta);

            // After thermalization, measure magnetization
            if (step >= thermalization) {
                double m = calculate_magnetization();
                total_mag += m;
                total_mag_sq += m * m;
            }
        }

        int steps_after_therm = total_steps - thermalization;   //We only want to consider the steps we took after thermalisation in our calculations
        double mag_avg = total_mag / steps_after_therm;         //FOr each value of beta we want to know the average magnetisation as we used our metropolis function
        double mag_sq_avg = total_mag_sq / steps_after_therm;
        double variance = mag_sq_avg - mag_avg * mag_avg;

        file << beta << "," << mag_avg << "," << mag_sq_avg << "," << variance << "\n";    //storing needed varibales
    }

    file.close();
}
```

Figure 5: Function to run the simulation and compute our final observables (Magnetization)

4 Results

4.1 Potts Model q=3

Table 1: q=3 Results (Every 5th Value)

Beta	M	M^2	Variance
0.5	0.045694	0.002764	0.000676
0.6	0.050622	0.003440	0.000877
0.7	0.059491	0.004724	0.001185
0.8	0.070512	0.006457	0.001485
0.9	0.101297	0.014089	0.003828
1.0	0.162533	0.034039	0.007622
1.1	0.801351	0.647424	0.005261
1.2	0.928021	0.861849	0.000626
1.3	0.961606	0.924922	0.000236
1.4	0.978502	0.957563	0.000096

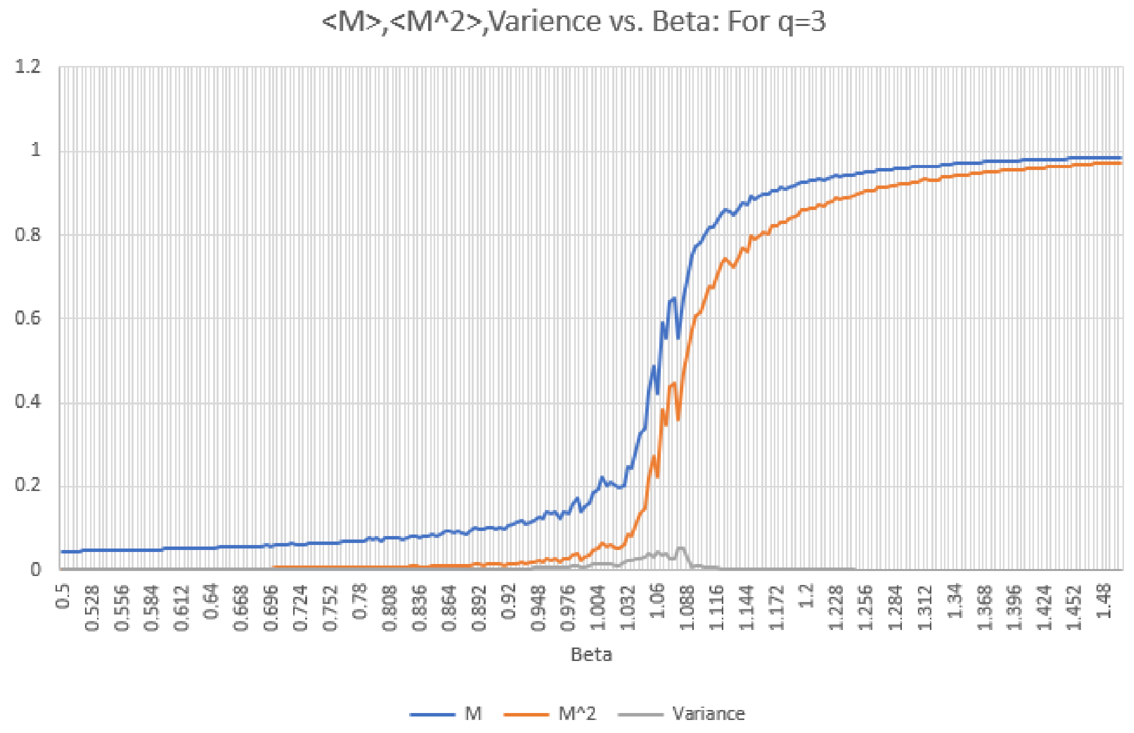


Figure 6: Phase Transition for q=3
The we a phase transition around beta 1.06.

4.2 Potts Model q=5

Table 2: Results q=5 (Every 5th Value)

Beta	**M**	**M^2**	**Variance**
0.5	0.034225	0.001399	0.000227
0.6	0.037100	0.001639	0.000263
0.7	0.039583	0.001882	0.000315
0.8	0.044062	0.002326	0.000385
0.9	0.049520	0.002936	0.000483
1.0	0.058485	0.004118	0.000698
1.1	0.079105	0.007559	0.001302
1.2	0.145279	0.030101	0.008995
1.3	0.894081	0.800526	0.001145
1.4	0.953266	0.908969	0.000252

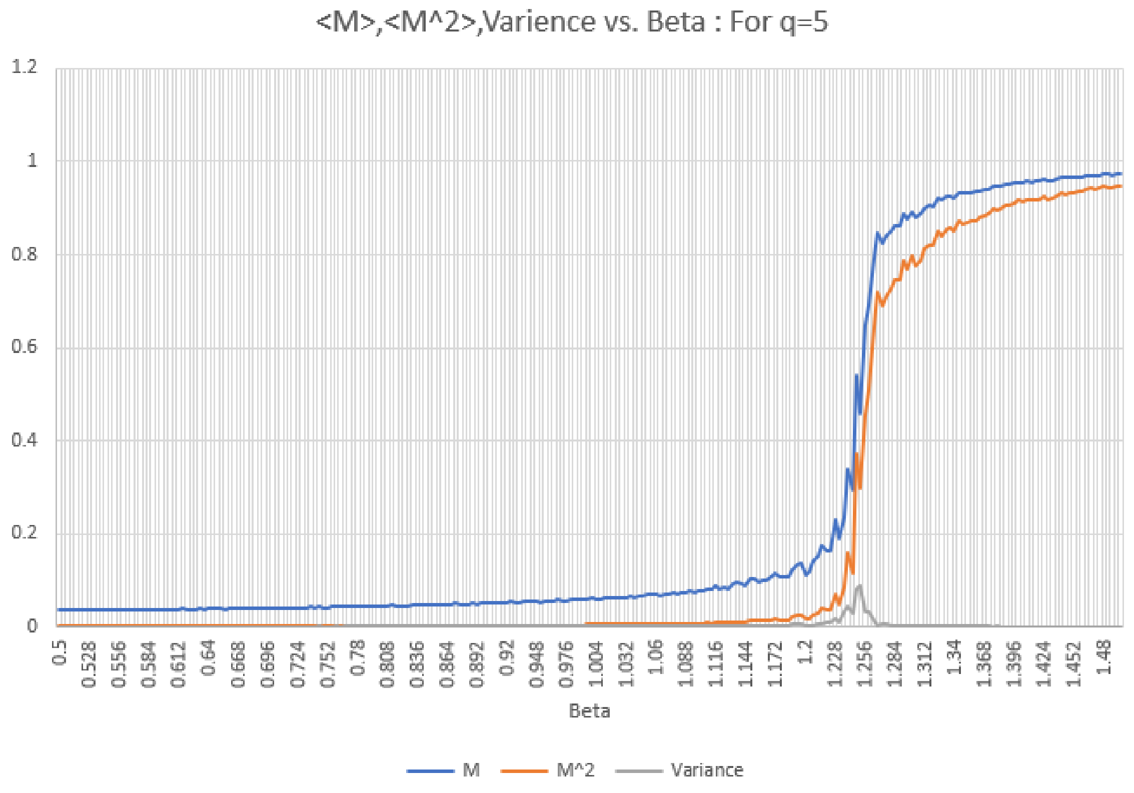


Figure 7: Phase transition for q=5
We see a critical beta value around 1.230. We see a much sharper transition then in q=3, this ii to be expected as it is a first order transition.