Statistical Physics Summer 2019

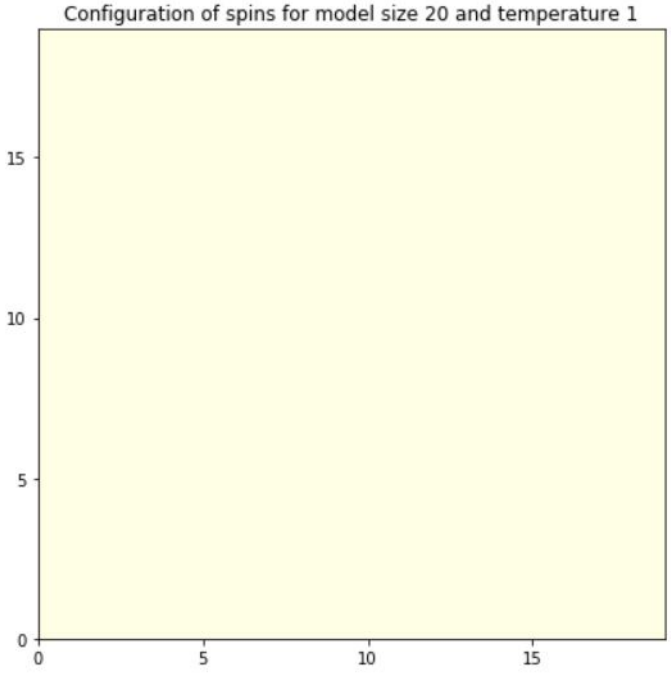
Task 3 - 2D Ising Model — 30.04, 2019

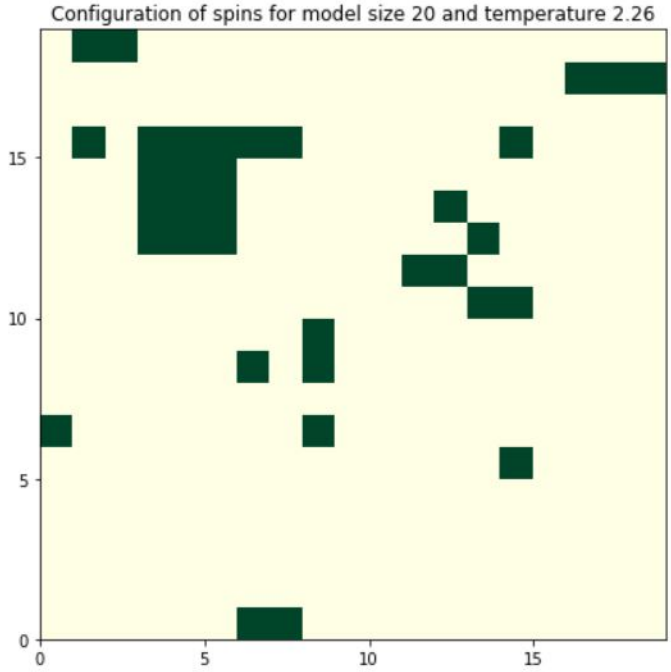
dr hab. inż. Grzegorz Pawlik *Author: Krzysztof Agieńczuk*

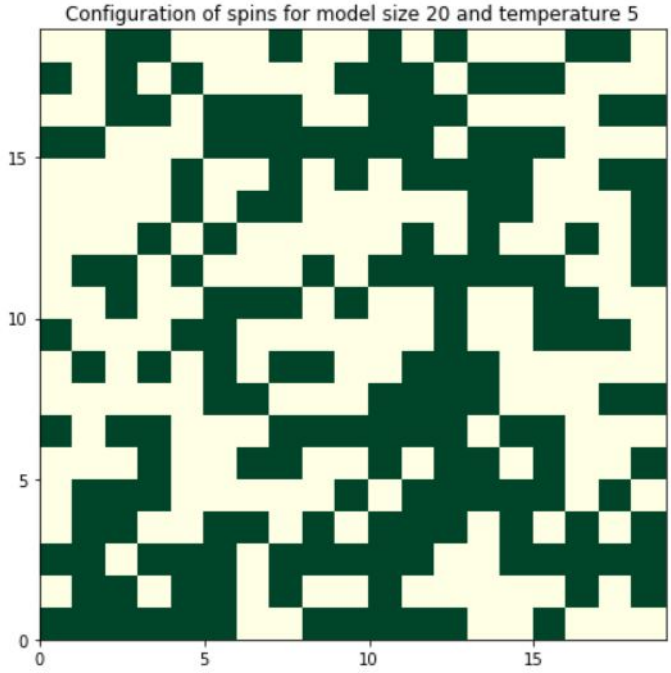
1 System behaviour

Note To improve readability of the document, all codes are placed in Appendix A.

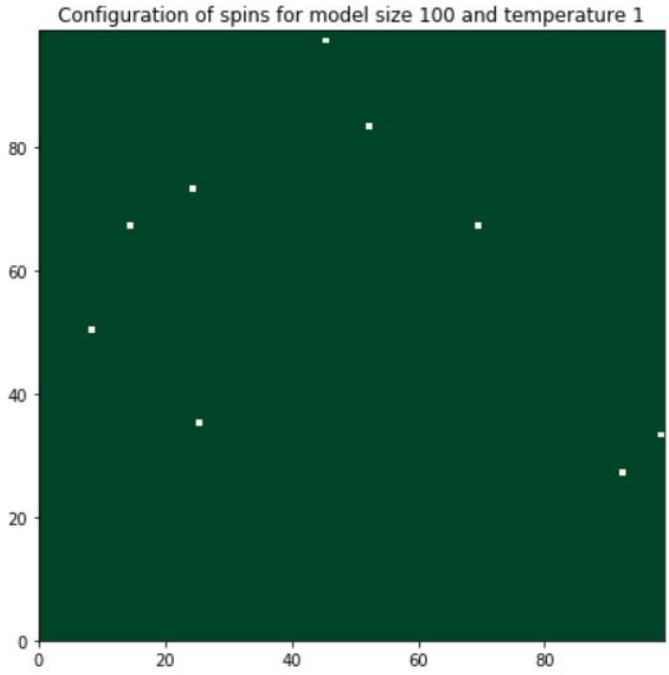
First task was to explore the behaviour of the system. Simulation was done for size `L=20 & L=100` and for temperature $T_C = 1$, $T_C = 2.26$ `&` $T_C = 5$. 30000 Monte Carlo Steps were used for cooling the system down.

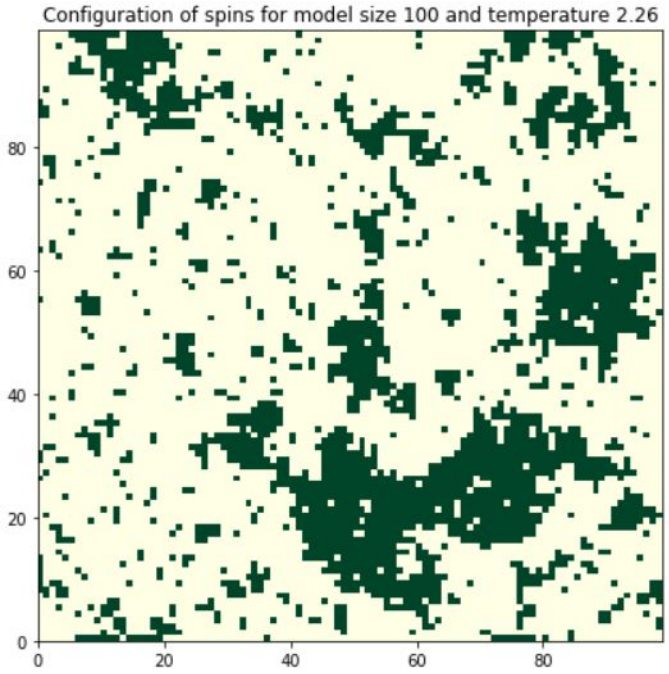


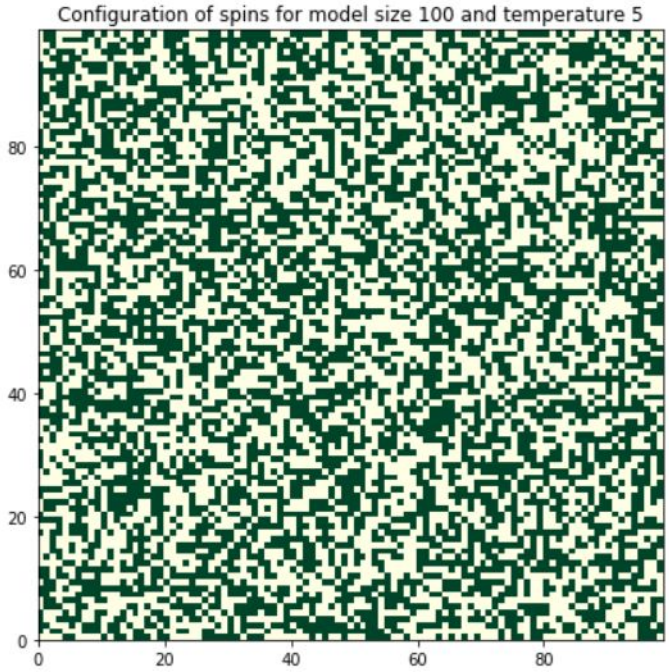

Configuration of spins for model size 20 and temperature 2.26
0
5
10
15
0
5
10
15


Configuration of spins for model size 20 and temperature 5
0
5
10
15
0
5
10
15

As seen above, for lower temperatures, small system cools down rather efficiently and quick. As we will see, it is not so efficient for larger systems. The higher the temperature and the bigger the size of the system, the longer it needs to cool down.





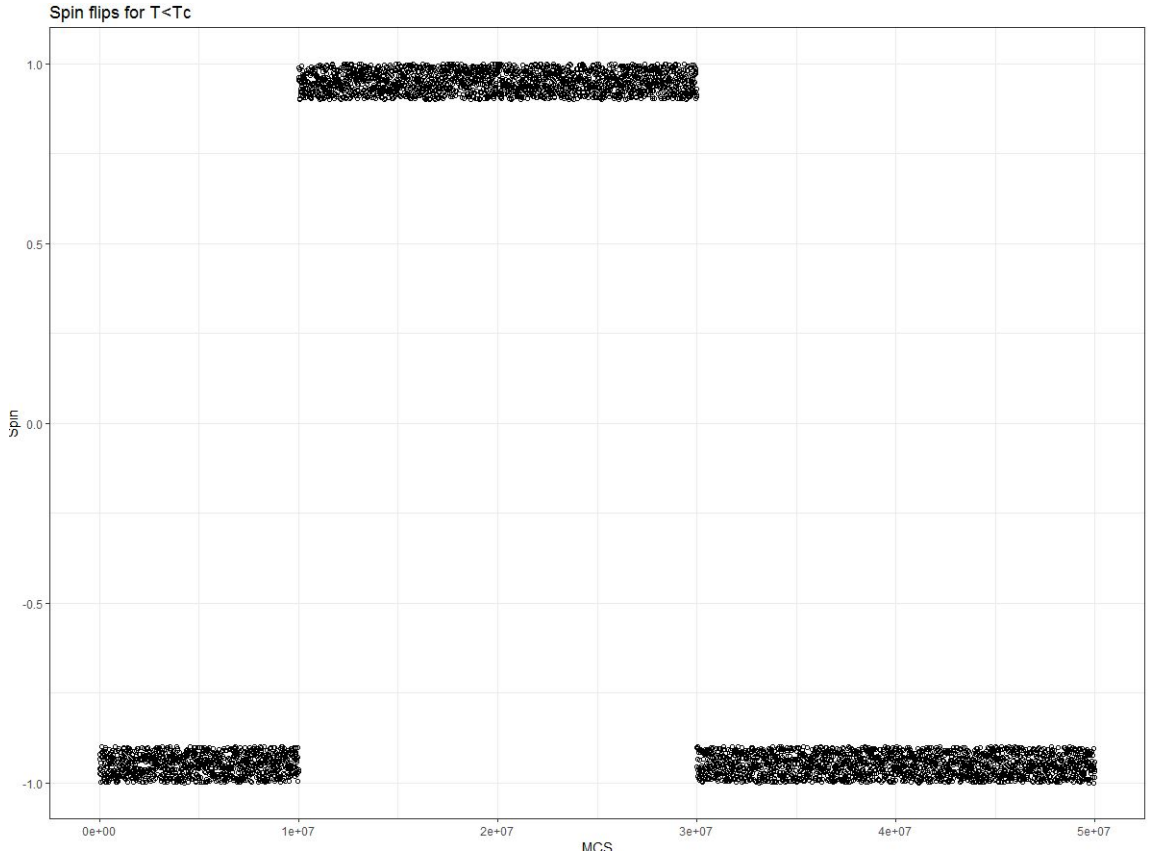


C++ was used for calculations. Code can be seen on listing 1.

Code for plotting is shown for only one of the plots, as it is highly repetitive. Initialisation of variables was skipped. As most of the plotting, this one was also done in Python. Unless specified otherwise, default programming languages were C++ for calculations and Python for plotting. Code can be seen in Appendix A, on listing 2

2 Flips between states

Below T_C system should spontaneously perform random flips in magnetisation from -1 to 1. This was observed only for huge number of MC steps. Every 1000th step was taken into account.



Spin flips for T=1.7 and size 20

Code for plotting is available on listing 3 and is available in Appendix A.

3 Temperature dependence on parameters

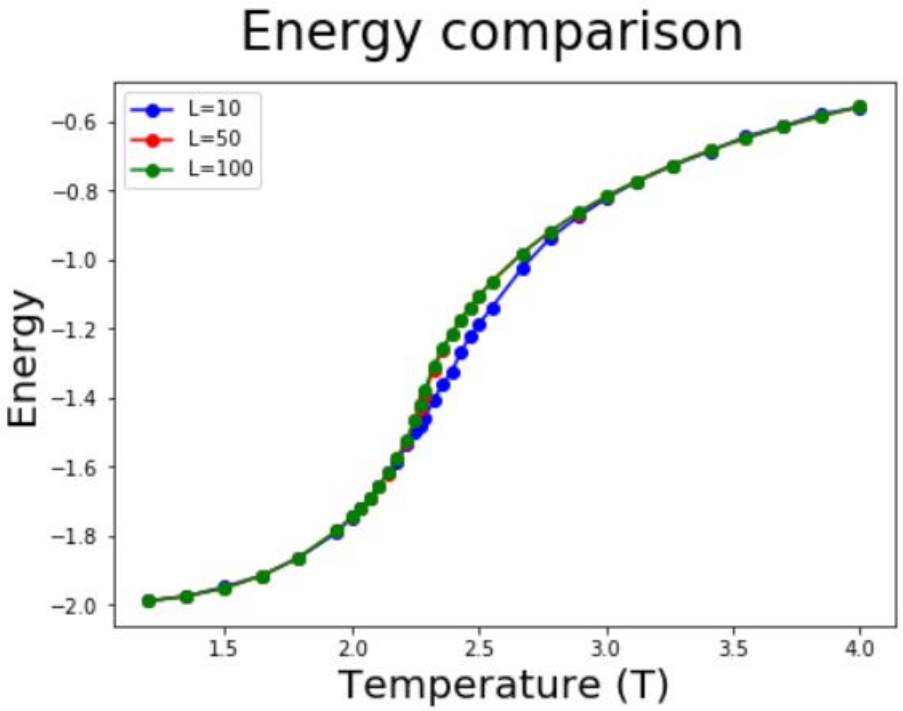
In this task, mean values of energy, magnetisation, susceptibility and specific heat were calculated. Also Binder's cumulant was used to determine T_C. Simulation lasted 230000 Monte Carlo Steps, including 30000 steps for cooling the system. Next every 1000th step was taken into account for calculations. Simulated lattices had sizes `L=10, L=50, L=100`.

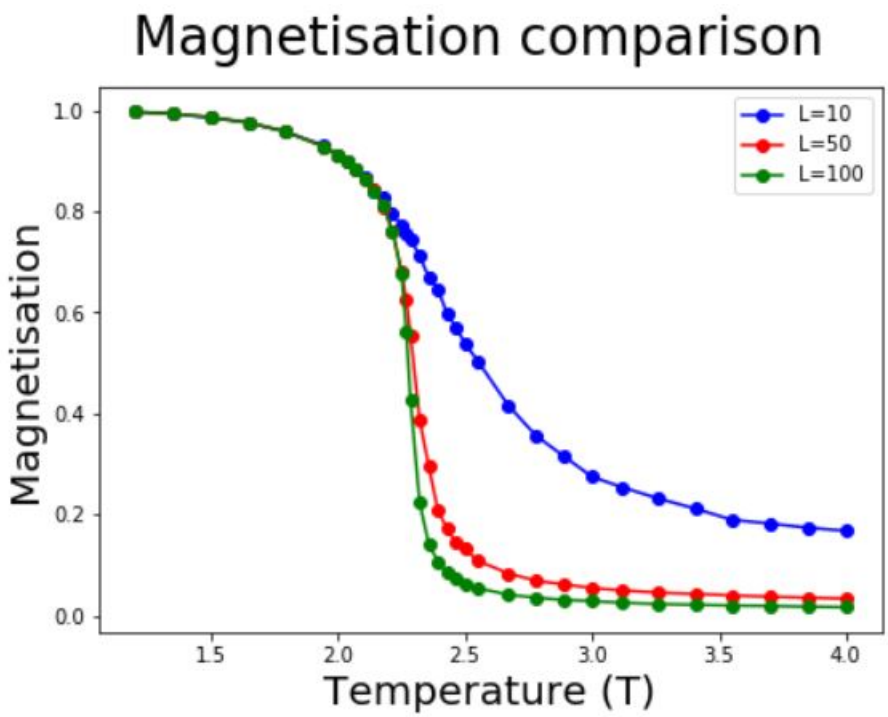
3.1 Corrections

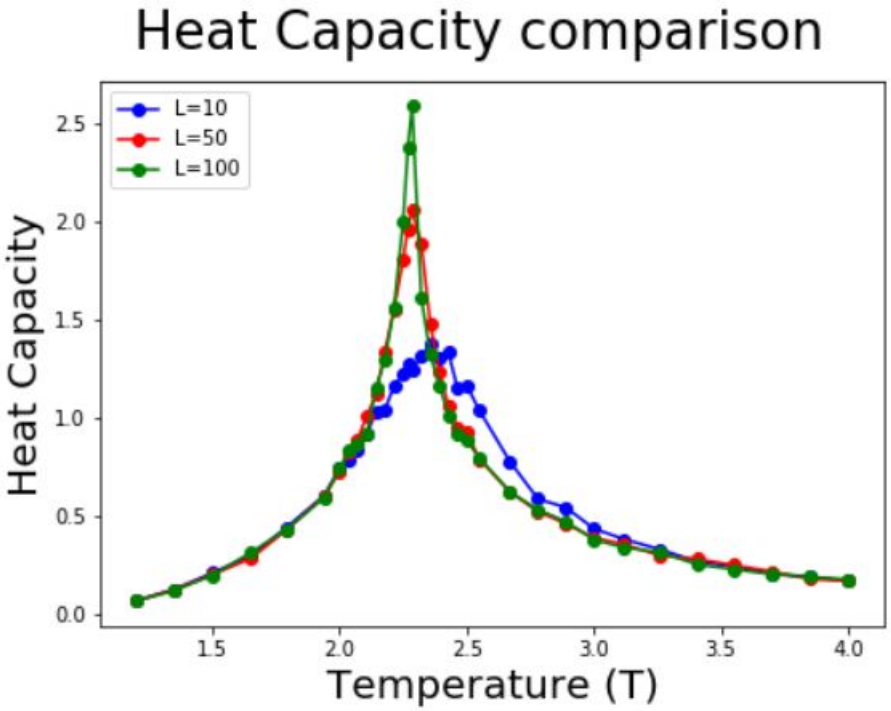
Mistake in previous code was connected to wrong averaging methods. Now, magnetisation is returned already divided by the size of the system. Returned values are summed up and the divided by amount of samples. Previously instead of dividing each sample by size of the system, only sum was divided by size times amount of samples (wrong order of calculations). Also amount of points was increased from 18 to 34. They were picked arbitrarily and can be seen on listing 4 (line 44).

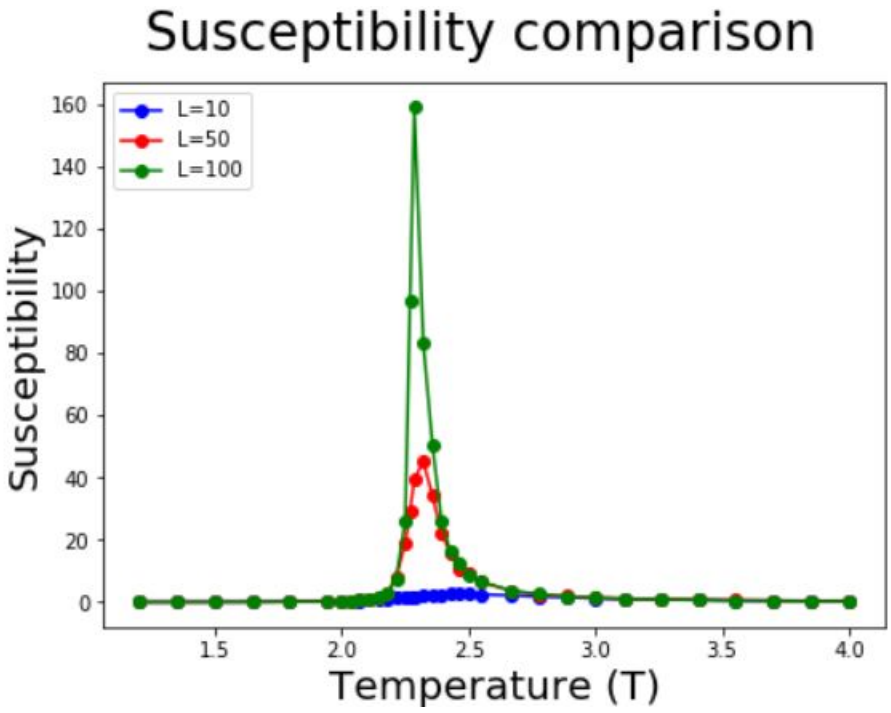
3.2 Refactoring

Using the occasion, small refactoring was introduced into the code. Now all parameters are counted in separate functions returning single values for provided parameters.




Magnetisation comparison
L=10
L=50
L=100
Magnetisation
Temperature (T)
1.0
0.8
0.6
0.4
0.2
0.0
1.5
2.0
2.5
3.0
3.5
4.0


Heat Capacity comparison
L=10
L=50
L=100
Heat Capacity
Temperature (T)
2.5
2.0
1.5
1.0
0.5
0.0
1.5
2.0
2.5
3.0
3.5
4.0

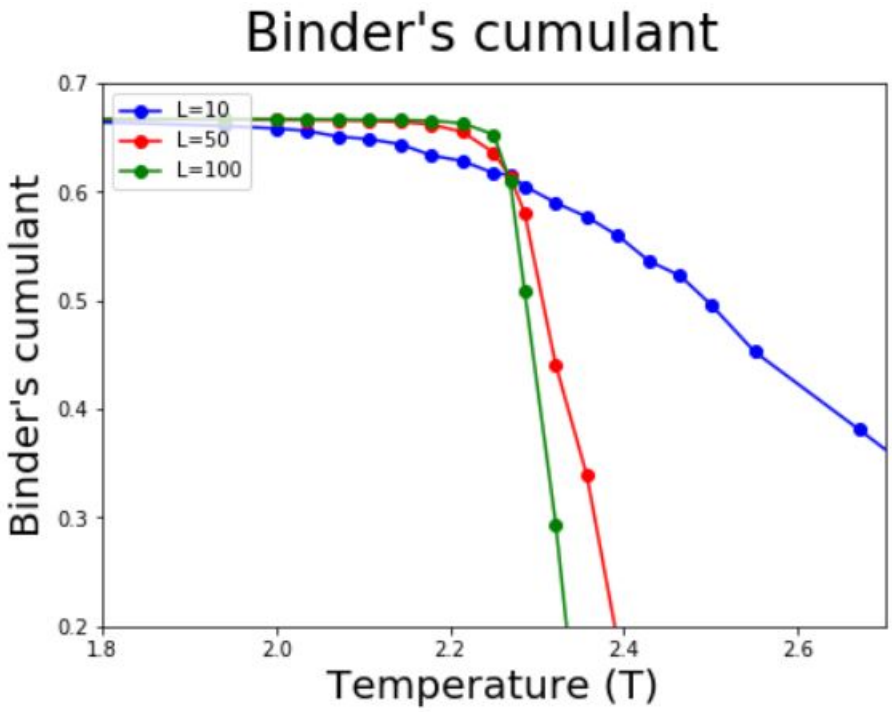


Code used for calculations was prepared in C++ and implemented in method testModel(). It is shown on listing 4

Code used for producing plots is presented on listing 5. Again, only one of plots is shown, due to repetitiveness of the code. Initialisation of variables is skipped.

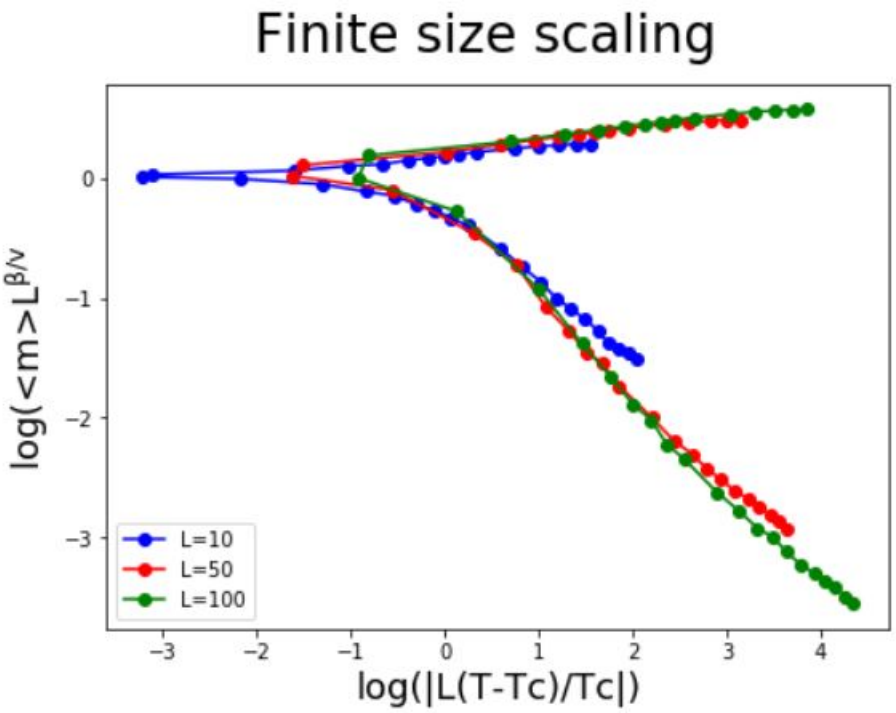
3.3 Binder's cumulant

To determine critical temperature, Binder's cumulant was plotted on one graph for all tested system sizes (10,50,100). The result is shown below. Large drop can be seen for bigger systems around temperature 2.25-2.26. This is the critical temperature.



Code used for plotting is analogical to the one available on listing 5

4 Finite size scaling



Code for calculating and plotting the finite size scaling is available on listing 7

5 Appendix A

Listing 1: "C++ code used for observing of magnetisation flips"

```
void countSpins(double arr[LATTICE_SIZE][LATTICE_SIZE]){
   std::fstream m;
   m.open("spins.txt");

   double M11 =0;
   double mag = 0;
   double tab[2000] = {0};
   int position = 0;
   for(int i=0; i<OMITTED_STEPS; i++) mcStep(arr, 1.9);

   for(int i=0; i<1; i++){
      for(int ii=0; ii<MONTE_CARLO_STEPS; ii++) {
         mcStep(arr, 1.9);
         if(ii % 1000 == 0){
            mag = calculateMagnetisation(arr);
            M11 = mag/(LATTICE_SIZE*LATTICE_SIZE);
            tab[position] = M11;
            position++;
         }
      }

   for(int ii=0; ii<2000; ii++) m<<tab[ii]<<",";
   std::cout<<i<<std::endl;
   }

   m.close();
}
```

Listing 2: "Python code for plotting behaviour of the system."

```
X, Y = np.meshgrid(range(20), range(20))

fig = plt.figure(figsize=(7,7))
ax1 = fig.add_subplot(111)

ax1.pcolormesh(X, Y, data, cmap=plt.cm.YlGn)
plt.title("Configuration of spins for model size 20 and temperature 5")
plt.xticks(np.arange(0, 20, 5.0))
plt.yticks(np.arange(0, 20, 5.0))
plt.show()
```

Listing 3: "R code for plotting spin flips"

```

```

```
dt <- read.table("spins.txt", header = F, sep = ",")
df <- data.frame(dt)
z = seq(from = 1, to = 50000000, by=10000)

ggplot() +
geom_point(data = df, aes(x=z, y=dt), shape = 1) +
ggtitle("Spin flips for T<Tc") +
xlab("MCS") +
ylab("Spin") +
ylim(-1,1) +
xlim(0,50000000) +
theme_bw()
```

Listing 4: "C++ code for main calculations"

```
double calculateEnergy(double arr[LATTICE_SIZE][LATTICE_SIZE]){
   double energy = 0;
   for(int i=0; i<LATTICE_SIZE; i++){
      for(int ii=0; ii<LATTICE_SIZE; ii++){
         double s = arr[i][ii];
         double nb = arr[mod(i+1, LATTICE_SIZE)][ii]
                 + arr[i][mod(ii+1, LATTICE_SIZE)]
                 + arr[mod(i-1, LATTICE_SIZE)][ii]
                 + arr[i][mod(ii-1, LATTICE_SIZE)];
         energy += -s*nb;
      }
   }
   //Two pairs and size
   return energy/(2*((LATTICE_SIZE*LATTICE_SIZE)));
}

double calculateMagnetisation(double arr[LATTICE_SIZE][LATTICE_SIZE]){
//average spin
   double sum=0;
   for(int i=0; i<LATTICE_SIZE; i++){
      for(int ii=0; ii<LATTICE_SIZE; ii++){
         sum += arr[i][ii];
      }
   }
   return sum/(LATTICE_SIZE*LATTICE_SIZE);
}

double calculateSusceptibility(double power2Magnetisation, double meanMagnetisation,
    double temperature){
   return ((LATTICE_SIZE*LATTICE_SIZE)*temperature)
   *(power2Magnetisation-meanMagnetisation*meanMagnetisation);
}

double calculateSpecificHeat(double power2Energy, double meanEnergy, double temperature){
   return (LATTICE_SIZE*LATTICE_SIZE)/(temperature*temperature)
   *(power2Energy-meanEnergy*meanEnergy);
```

```
}

double calculateBindersCumulant(double power4Spins, double power2Spins) {
   return 1 - (power4Spins/(3*power2Spins*power2Spins));
}

void testModel(double lattice[LATTICE_SIZE][LATTICE_SIZE]){
   // Prepare time series
   double T[TEMP_POINTS] = {1.2,1.35, 1.5, 1.65, 1.79, 1.94, 2., 2.03571429, 2.07142857, 2.10714286, 2.14285714,
      2.17857143, 2.21428571, 2.25, 2.269, 2.28571429, 2.32142857, 2.35714286, 2.39285714,
      2.42857143, 2.46428571, 2.5, 2.55, 2.67, 2.78, 2.89, 3., 3.12, 3.26, 3.41, 3.55, 3.7, 3.85, 4.};

   // Prepare helper variables
   double iT=0, mag=0, ene=0, sp=0;
   double meanEnergy=0, meanMagnetisation=0, susceptibility=0, specificHeat=0, bindersCumulant=0;
   double power2Energy=0, power2Magnetisation=0, power2Spins=0, power4Spins=0;

   // Start model
   for(int point=0; point<TEMP_POINTS; point++){
      meanMagnetisation = 0;
      meanEnergy = 0;
      power2Energy = 0;
      power2Magnetisation = 0;
      power2Spins=0;
      power4Spins=0;
      susceptibility=0;
      specificHeat=0;
      bindersCumulant=0;

      iT = 1/T[point];

      // Omit steps
      for(int i=0; i<OMITTED_STEPS; i++) mcStep(lattice, iT);

      // "Real" calculation
      for(int ii=0; ii<MONTE_CARLO_STEPS; ii++){
         mcStep(lattice, iT);

         if(ii%factor == 0){
         mag = calculateMagnetisation(lattice);
         ene = calculateEnergy(lattice);
         sp = sumSpins(lattice);
         meanMagnetisation += abs(mag);
         meanEnergy += ene;
         power2Energy += ene*ene;
         power2Magnetisation += mag*mag;
         power2Spins += sp*sp;
         power4Spins += sp*sp*sp*sp;
         }
      }
```

```
86
87
88      meanMagnetisation = meanMagnetisation/(MONTE_CARLO_STEPS/factor);
89      meanEnergy = meanEnergy/(MONTE_CARLO_STEPS/factor);
90
91      // Average
92      power2Energy = power2Energy/(MONTE_CARLO_STEPS/factor);
93      power2Magnetisation = power2Magnetisation/(MONTE_CARLO_STEPS/factor);
94      power2Spins = power2Spins/(MONTE_CARLO_STEPS/factor);
95      power4Spins = power4Spins/(MONTE_CARLO_STEPS/factor);
96
97      susceptibility = calculateSusceptibility(power2Magnetisation, meanMagnetisation, iT);
98      specificHeat = calculateSpecificHeat(power2Energy, meanEnergy, 1/iT);
99      bindersCumulant = calculateBindersCumulant(power4Spins, power2Spins);
100
101     E[point] = meanEnergy;
102     M[point] = meanMagnetisation;
103     X[point] = susceptibility;
104     C[point] = specificHeat;
105     U[point] = bindersCumulant;
106     }
107
108     writeToFile();
109 }
```

Listing 5: "Python plot for plotting parameters (specifically magnetisation)"

```
1  fig = plt.figure(figsize=(7, 5))
2  fig.suptitle("Binder's cumulant for sparse points", fontsize = 26)
3  ax1 = fig.add_subplot(111)
4  axes = plt.gca()
5  axes.set_ylim([0.2,0.7])
6
7  ax1.plot(T, M10, c='b', marker="o", label='L=10')
8  ax1.plot(T, M50, c='r', marker="o", label='L=50')
9  ax1.plot(T, M100, c='g', marker="o", label='L=100')
10 plt.xlabel("Temperature (T)", fontsize=20);
11 plt.ylabel("Binder's cumulant", fontsize=20)
12
13 plt.legend(loc='upper left');
14 plt.show()
```

Listing 6: "C++ code for Monte Carlo Step"

```
1  void mcStep(double arr[LATTICE_SIZE][LATTICE_SIZE], double beta){
2     for(int i=0; i<LATTICE_SIZE; i++){
3        for(int ii=0; ii<LATTICE_SIZE; ii++){
4           int a = rand() % LATTICE_SIZE;
5           int b = rand() % LATTICE_SIZE;
```

```
        double s = arr[a][b];
        double nb = arr[mod(a+1, LATTICE_SIZE)][b]
                + arr[a][mod(b+1, LATTICE_SIZE)]
                + arr[mod(a-1, LATTICE_SIZE)][b]
                + arr[a][mod(b-1, LATTICE_SIZE)];
        double cost = 2*s*nb;
        double r = (((double) rand() / (RAND_MAX)));
        if (cost<0) s *= -1;
        else if ( r < exp(-cost*beta)) s*= -1;
        arr[a][b] = s;
     }
   }
}
```

Listing 7: "Python code for plotting finite size scaling behavioue"

```
nt = 34 #number of temperature points
fig = plt.figure(figsize=(7, 5))
fig.suptitle("Finite size scaling", fontsize = 26)
ax1 = fig.add_subplot(111)

for i in range(nt):
Tfinite10[i] = abs(((T[i]-2.26)/2.26)*10)

for i in range(nt):
Tfinite50[i] = abs(((T[i]-2.26)/2.26)*50)

for i in range(nt):
Tfinite100[i] = abs(((T[i]-2.26)/2.26)*100)

for i in range(len(M10)):
m10[i] = M10[i]*(np.power(10,1/8))

for i in range(len(M50)):
m50[i] = M50[i]*(np.power(50,1/8))

for i in range(len(M100)):
m100[i] = M100[i]*(np.power(100,1/8))

ax1.plot(np.log((Tfinite10)), np.log(m10), c='b', marker="o", label='L=10')
ax1.plot(np.log((Tfinite50)), np.log(m50), c='r', marker="o", label='L=50')
ax1.plot(np.log((Tfinite100)), np.log(m100), c='g', marker="o", label='L=100')
ax1.set_ylabel(r'log(<m>$\mathregular{L^{\beta/v}}$', fontsize=18)
ax1.set_xlabel('log(|L(T-Tc)/Tc|)', fontsize=18)

plt.legend(loc='lower left');
plt.show()
```