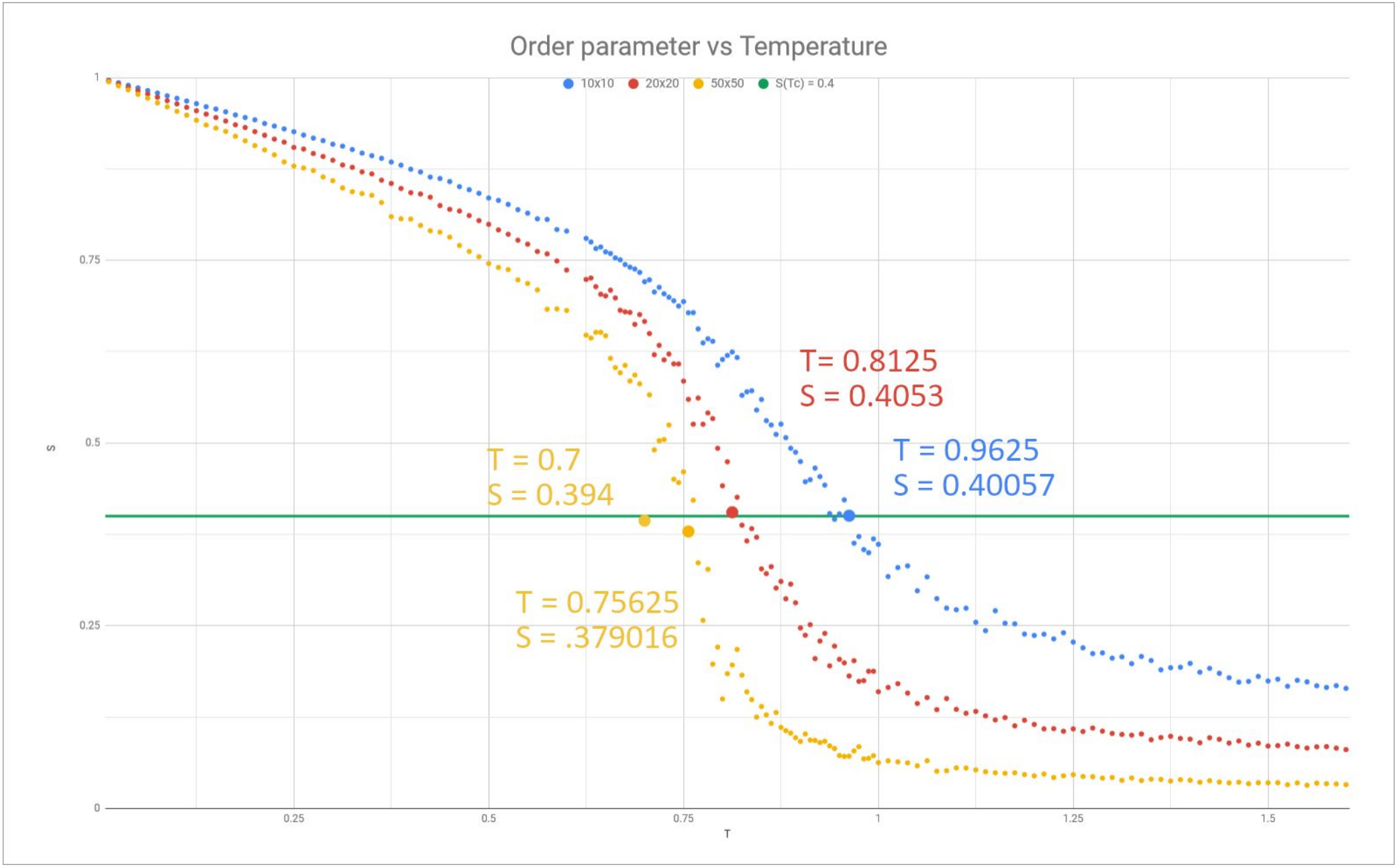
Jakub Stawik
218715

Nematic Liquid Crystal

Parameters used for the simulation:
For every temperature a new lattice was created and 30 000 Monte Carlo steps were skipped to move to equilibrium state. Then 1 500 configurations were recorded, 200 steps in between each, and averaged to obtain order parameter for the temperature.
3 lattice sizes were considered: 10x10, 20x20 and 50x50.

Parameters	
Lattice sizes:	10, 20, 50
MCS to equilibrium	30000
Gap between configurations	200
Configurations for single datapoint	1500
Delta parameter (for trial configuration)	10
Temperatures (157 datapoints)	[1/80 : 1/80 : 0.6] (low range) [0.6 : 1/160 : 1] (mid range, high detail) [1 : 1/80 : 1.6] (high range)
Execution times	10x10: 17min 20x20: 1h 8min 50x50: 7h 11min


Order parameter vs Temperature
10x10
20x20
50x50
S(Tc) = 0.4
T= 0.8125
S = 0.4053
T = 0.9625
S = 0.40057
T = 0.7
S = 0.394
T = 0.75625
S = .379016
S
T
1
0.75
0.5
0.25
0
0.25
0.5
0.75
1
1.25
1.5

Main engine code:

```
#include <iostream>
#include <cmath>
#include <stdlib.h>
#include <cstdio>
#include <ctime>
#define PI 3.14159265
using namespace std;

#define L 50
int LATTICE [L][L];
int SKIPSTEPS = 30000;
int CONFIGS = 1500;
int DELTA = 10;

double calculate_energy(int trial, int n1, int n2, int n3, int n4){
   return  -1.5*(pow(cos(PI*(trial-n1)/180), 2)
                +pow(cos(PI*(trial-n2)/180), 2)
                +pow(cos(PI*(trial-n3)/180), 2)
                +pow(cos(PI*(trial-n4)/180), 2))+2;
}

void MCS(int steps, double T, int delta){
   int trial;
   double energy;
   double energy0 = 0;
   double diff;
   for(int i = 0; i < steps; i++){
       for(int x = 0; x < L; x++){
           for(int y = 0; y < L; y++){
               //Single MCS
               //CURRENT ENERGY
               if(x == 0 and y ==0) energy0 = calculate_energy(LATTICE[x][y], LATTICE[((x+1)%L)][y]
                           , LATTICE[L-1][y]
                           , LATTICE[x][((y+1)%L)]
                           , LATTICE[x][L-1]);
               else if(x == 0) energy0 = calculate_energy(LATTICE[x][y], LATTICE[((x+1)%L)][y]
                           , LATTICE[L-1][y]
                           , LATTICE[x][((y+1)%L)]
                           , LATTICE[x][((y-1)%L)]);
               else if(y == 0) energy0 = calculate_energy(LATTICE[x][y], LATTICE[((x+1)%L)][y]
                           , LATTICE[((x-1)%L)][y]
                           , LATTICE[x][((y+1)%L)]
                           , LATTICE[x][L-1]);
               else energy0 = calculate_energy(LATTICE[x][y], LATTICE[((x+1)%L)][y]
                           , LATTICE[((x-1)%L)][y]
                           , LATTICE[x][((y+1)%L)]
                           , LATTICE[x][((y-1)%L)]);
               //TRIAL ENERGY
               trial = LATTICE[x][y] + round((rand()%1000/1000.0-0.5)*delta);
               if(trial < -90) trial += 180;
               if(trial > 90) trial -= 180;
               if(x == 0 and y ==0) energy = calculate_energy(trial, LATTICE[((x+1)%L)][y]
                           , LATTICE[L-1][y]
                           , LATTICE[x][((y+1)%L)]
                           , LATTICE[x][L-1]);
               else if(x == 0) energy = calculate_energy(trial, LATTICE[((x+1)%L)][y]
                           , LATTICE[L-1][y]
                           , LATTICE[x][((y+1)%L)]
                           , LATTICE[x][((y-1)%L)]);
               else if(y == 0) energy = calculate_energy(trial, LATTICE[((x+1)%L)][y]
                           , LATTICE[((x-1)%L)][y]
                           , LATTICE[x][((y+1)%L)]
                           , LATTICE[x][L-1]);
               else energy = calculate_energy(trial, LATTICE[((x+1)%L)][y]
                           , LATTICE[((x-1)%L)][y]
                           , LATTICE[x][((y+1)%L)]
                           , LATTICE[x][((y-1)%L)]);
               //FLIPPING
               diff = energy - energy0;
               if(diff < 0) LATTICE[x][y] = trial;
               else if(rand()%1000/1000.0 < exp(-diff/T)) LATTICE[x][y] = trial;
           }
       }
   }
}
```

```
double calculate_Qxx(){
   double Qxx = 0.0;
   for(int i = 0; i<L; i++)
       for(int j = 0; j<L; j++) Qxx += 2 * pow(cos(PI*LATTICE[i][j]/180), 2) - 1;
   return Qxx/pow(L, 2);
}
double calculate_Qxy(){
   double Qxy = 0.0;
   for(int i = 0; i<L; i++)
       for(int j = 0; j<L; j++) Qxy += 2 * cos(PI*LATTICE[i][j]/180) * sin(PI*LATTICE[i][j]/180);
   return Qxy/pow(L, 2);
}

double calculate_S(){
   double Qxx = calculate_Qxx();
   double Qxy = calculate_Qxy();
   return sqrt(pow(Qxx, 2) + pow(Qxy, 2));
}
int main(int argc, char *argv[]) {
   // Temperature read
   string arg = argv[1];
   double T = atof(arg.c_str());
   for(int i = 0; i < L; i++){
       for(int j = 0; j < L; j++) LATTICE[i][j] = 0;
   }

   srand(time(NULL));
   // Skip to stable
   MCS(SKIPSTEPS, T, DELTA);
   // Calculations
   double S = 0.0;
   for(int i = 0; i < CONFIGS; i++){
       MCS(200, T, DELTA);
       S += calculate_S();
   }
   S = S/CONFIGS;
   cout<<T<<'\t'<<S<<'\n';
   return 0;
}
```

Python driver running engines in parallel:

```
from multiprocessing import Pool
import os
import numpy as np
import time


def drive(i):
    os.system(f'crystals.exe {i}')

if __name__ == '__main__':
   start_time = time.time()
   temps = [i/80 for i in range(1, 49)] + [i/160 for i in range(100, 160)] + [i/80 for i in range(80,129)]
   pool = Pool()
   pool.map(drive, temps)
   pool.close()
   pool.join()
   print("--- %s seconds ---" % (time.time() - start_time))
```

Both can be downloaded @
https://github.com/jstawik/WUST-BDA/tree/master/Statistical_physics/Crystals