Homework 5
Monte Carlo Simulation

Yuding Ai
Penn ID: 31295008
MSE 561 - Atomic Modeling in Materials Science

April 21, 2017

1 Metropolis type of Monte Carlo simulation

In this work, we will employ the Metropolis type of Monte Carlo simulation and assume that the kinetic energy depends on velocities only and potential energy depends on the positions only. In such way that the movement acceptance probability could be described as the following:

$$p^{mov}(i \rightarrow j) = min[1, e^{-(PE_j - PE_i)/k_B T}] \tag{1}$$

Using this acceptance probability, we could then implement our Monte Carlo simulation following the flowchart as depicted in Figure 1.

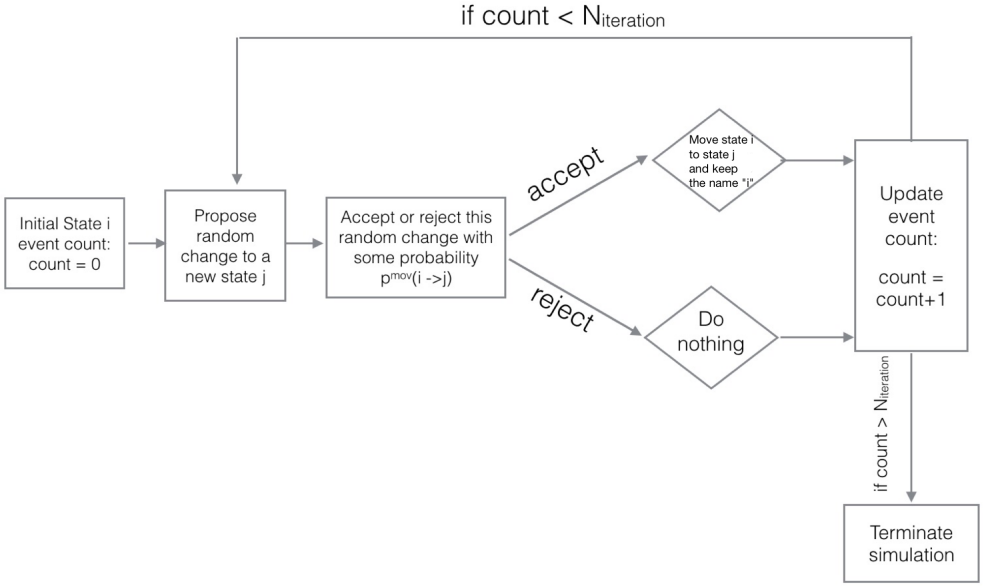


Figure 1: Flowchart for Monte Carlo Simulation

1.1 Structural relaxation

As to adjust the configuration of the block of atoms corresponding to different temperatures, in each Monte Carlo step, we propose a new state by **randomly pick** an atom from the block and then propose a new position following the prescription:

$$\begin{aligned} x_j &= x_i + \alpha\zeta_x \\ y_j &= y_i + \alpha\zeta_y \end{aligned} \tag{2}$$

where ζ_x and ζ_y are **randomly generated parameters** in between -1 and $+1$ and α is the maximum displacement allowed.
Experiment has shown that to get the fastest convergence, the choice of α should lead the movement acceptance probability to be about 0.2 to 0.4. Due to this reason, we will adjust the α value on the fly as our Monte Carlo simulation proceeds.

1.2 Pseudo code for our Monte Carlo simulation

Armed with the above knowledge, we could then summarize our Monte Carlo simulation into Algorithm 1 as a pseudo code. (see detailed implementation at Appendix, 2dblock.cpp, method: MonteCarlo()) Lastly, after we reached the equilibrium state for each temperature, we run an additional Monte carlo simulation with fewer steps (since it has reached the equilibrium already) as to get the $< PE >$ and $< Total\ hydrostatic\ pressure >$.

Algorithm 1 Monte Carlo simulation with various Temperatures

1: Set the initial configuration and fix T
2: **while 500000 iterations do**
3: Attempt to move the atoms following Equation 2
4: **Update the neighbor list** (since we moved the atoms in previous step)
5: Calculate the new **Potential Energy** of the current configuration
6: Calculate the movement acceptance probability p^{mov} using Equation 1
7: Generate a **random number** ξ between 0 and 1 then compare it to p^{mov}:
8: **if** $p^{mov} \geq \xi$ **then**
9: Accept the movement
10: **else**
11: Retain the old configuration
12: Check the average value of p^{mov} over 1000 steps and then adjust α to let p^{mov} lands in between 0.2 and 0.4
13: Calculate and record the hydrostatic pressure for the final configuration
14: Plot the final configuration and Radial distribution function
15: Plot PE vs Iterations and p^{mov} vs Iterations.

2 Simulation result

The output of the Monte Carlo simulation for Temperatures from 5K to 80K is demonstrated through Figure 2 to Figure 9 and for each temperature, we attached the Configuration plot along with hydrostatic pressure on each atom, the RDF distribution and the Potential energy vs Iteration plot. From 5K to 40K, we provide an additional movement acceptance probability p^{mov} vs Iteration plot and for temperatures from 45K to 80K, we provide an additional Zoom-in version of configuration plot.

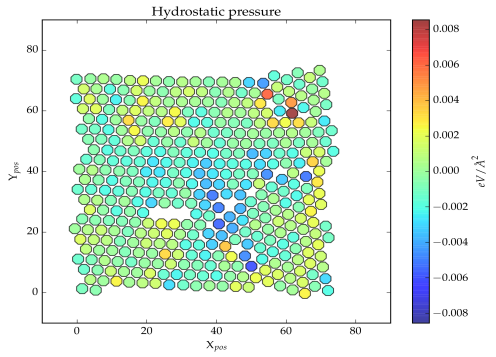
First of all, we start with the 'square' block from Assignment 2 and perform a relaxation simulation by fixing the temperature at 5K. Our experiments shows that one need at least **500000** iterations to reach the equilibrium and as we could see from Figure 2 (d), the Potential energy keep decreasing gradually and finally reached at about -11.74 eV. The plot of movement acceptance probability VS Iterations in Figure 2 (c) shows that we have a good control of α values such that our p^{mov} is always in between 0.2 and 0.4. Further, the RDF plot at Figure 2 (b) shows a clear 7 peak distribution which indicates we have a crystalline structure and finally, the configuration plot agrees with a crystalline structure as well.

Next, we run 15 separate Monte Carlo simulation as to increase the temperature from 5K to 80K. Even though, not all the simulation needs 500000 iterations to reach its equilibrium, for example, for 10K this is overkill, but when it is at the phase transformation state, such as 55K, it is necessary. Therefore, as for consistency, **we fix the running time for all simulation to be 500000 iterations**.

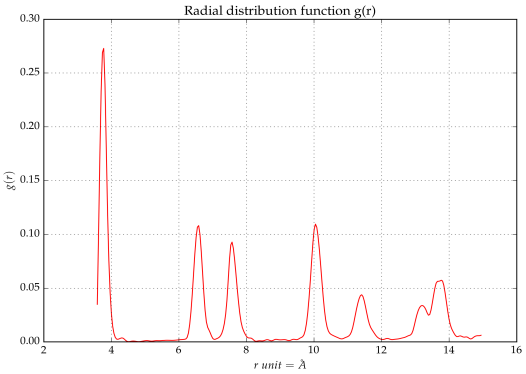
As is shown in the output plots, we see that from 5K to 30K, the configuration of the atoms has not changed much. Although the radial distribution function became more and more fuzzy and blur as the temperature increased, we can still see 6 peaks such that **upon till 35K, the material is till in a crystalline structure**. After 35K, the RDF plot loss its 6 peak distribution and the potential energy starts to increase dramatically and these are signs indicating that the phase transformation from solid to liquid is about to happen. Then **from 40K to 60K, the phase transformation took place and after 60K, according to the radial distribution function, we could only see one big peak left thus indicates that the crystalline structure has broken and the material is in a liquid phase. Lastly from 70K to 80K**, in both the final configuration plot and potential energy plot, we see the atoms are sparsely distributed (volume increased a lot from the initial 80 x 80 $\mathring{A}^2$ block to the final 900 x 900 $\mathring{A}^2$ block), and the potential energy once again starts to increase dramatically hence indicates another phase transformation from liquid to gas is taking place. Therefore, **those atom are possibly formed a gas phase of the material**.

Last, we plot the Energy vs Temperatures, Total hydrostatic pressure vs Temperatures and Heat capacity vs Temperatures at Figure 10. From these plots, we see that **The Total Energy increase gradually as temperature increases and when phase transition took place, the rate of such increment became faster.** (although not as steep as a jump, due to the fact that our model size is pretty small) On the other hand, as is expected, **the Heat capacity plot shows two peaks indicating phase transformations, one for solid to liquid and another**

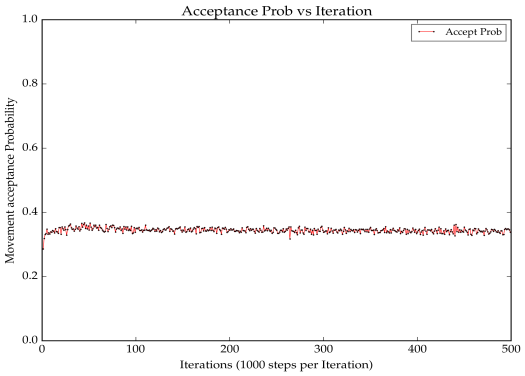
for liquid to gas (both are not very sharp, due to the same reason of the model size). Last, the pressure vs Temperature plot shows that the pressure start with about 0 and then there is a increment of magnitude as the block of atoms goes toward to phase transformation and then finally when the block reaches the final gas state, the pressure goes back to zero, which is expected.



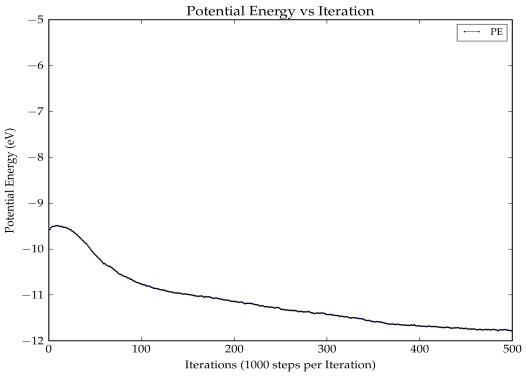
(a) T = 5K, Final Configuration and $P_{hydrostatic}$



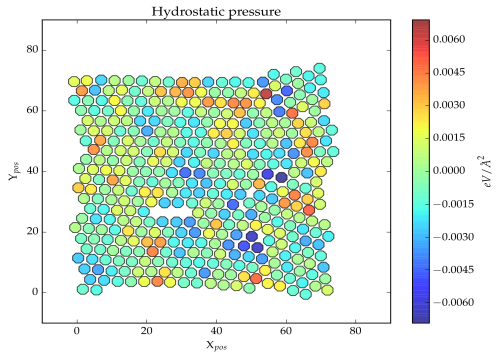
(b) T = 5K, RDF



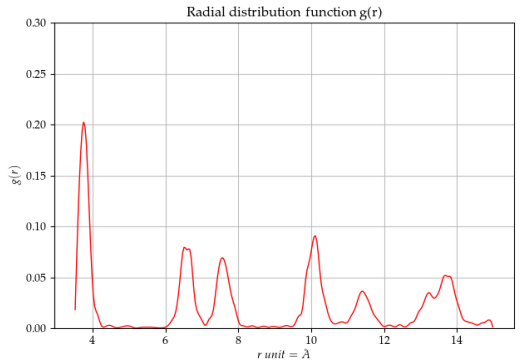
(c) T = 5K, Acceptance probability



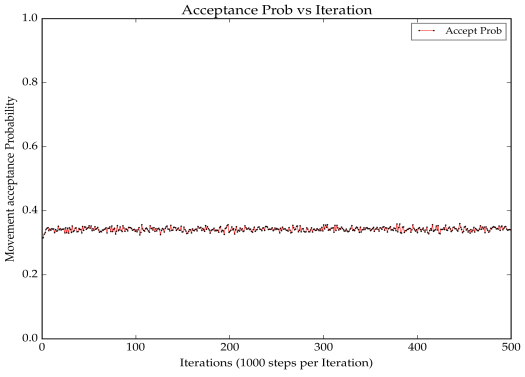
(d) T = 5K, PE vs iteration



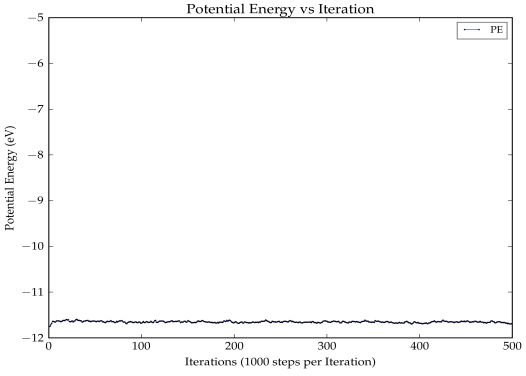
(e) T = 10K, Final Configuration and $P_{hydrostatic}$



(f) T = 10K, RDF

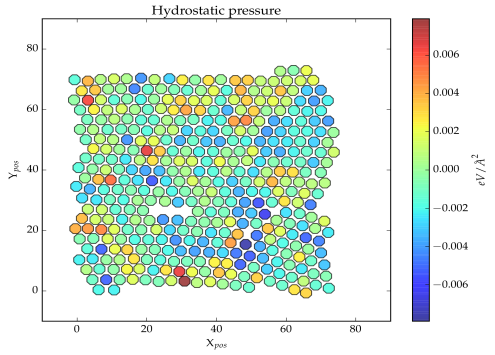


(g) T = 10K, Acceptance probability

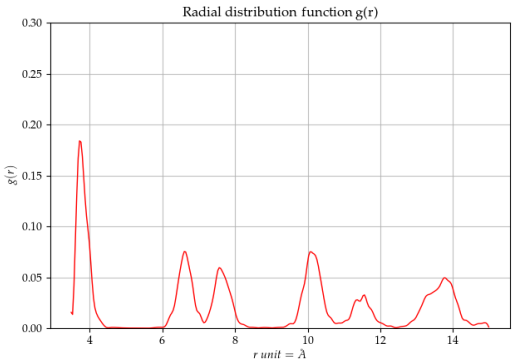


(h) T = 10K, PE vs iteration

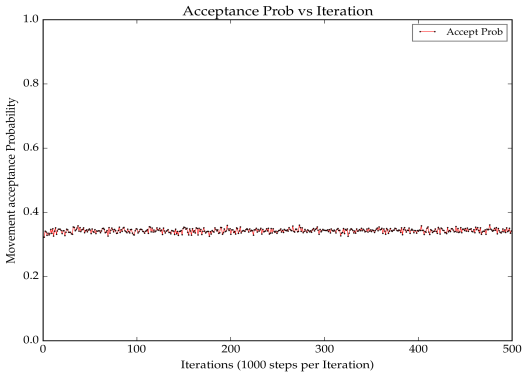
Figure 2: Temperature at 5K and 10K



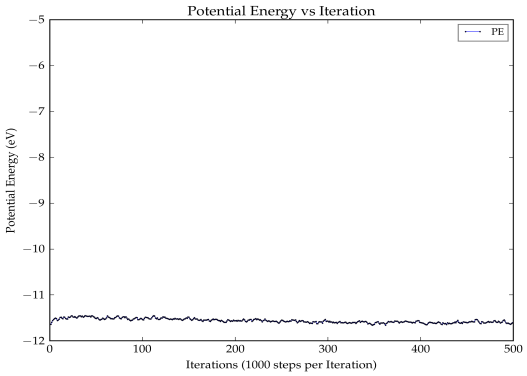
(a) T = 15K, Final Configuration and $P_{hydrostatic}$



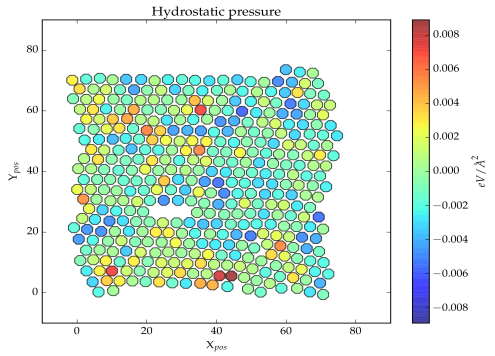
(b) T = 15K, RDF



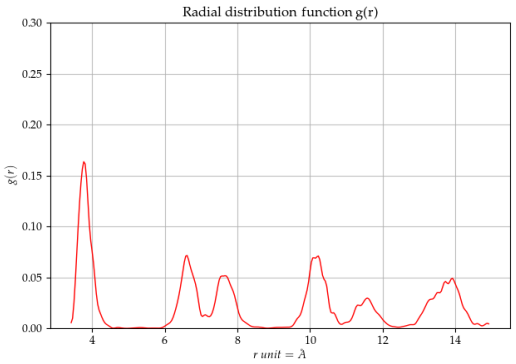
(c) T = 15K, Acceptance probability



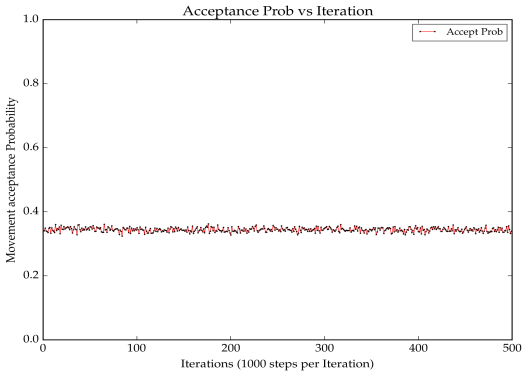
(d) T = 15K, PE vs iteration



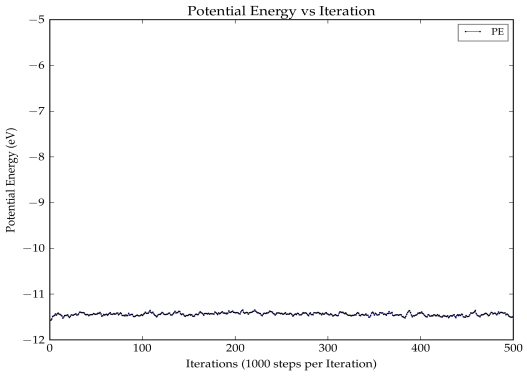
(e) T = 20K, Final Configuration and $P_{hydrostatic}$



(f) T = 20K, RDF

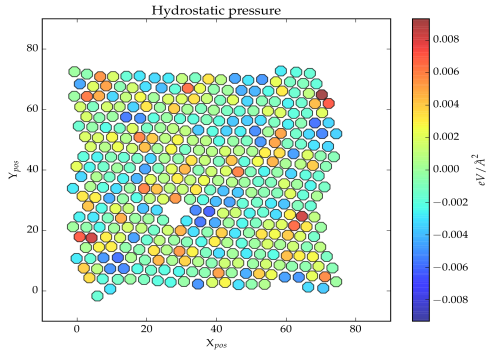


(g) T = 20K, Acceptance probability

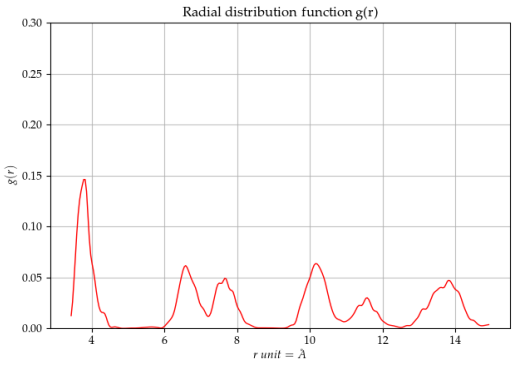


(h) T = 20K, PE vs iteration

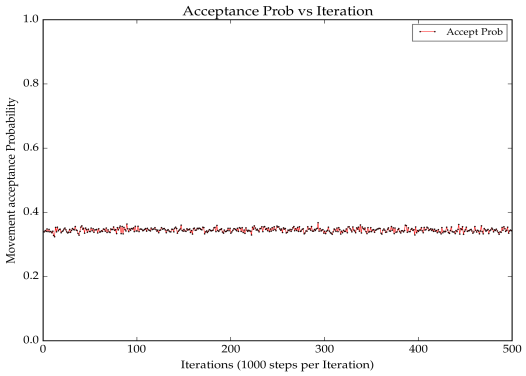
Figure 3: Temperature at 15K and 20K



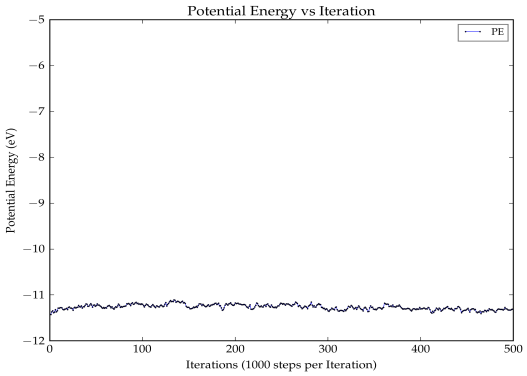
(a) T = 25K, Final Configuration and $P_{hydrostatic}$



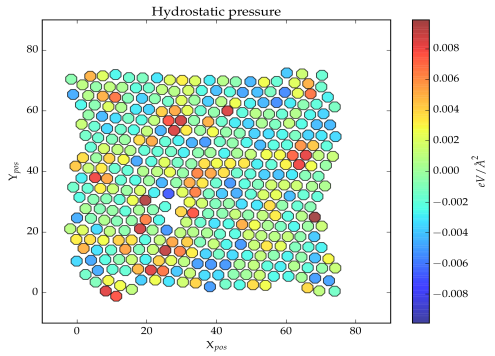
(b) T = 25K, RDF



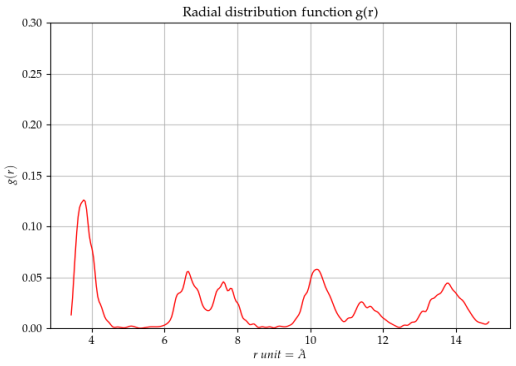
(c) T = 25K, Acceptance probability



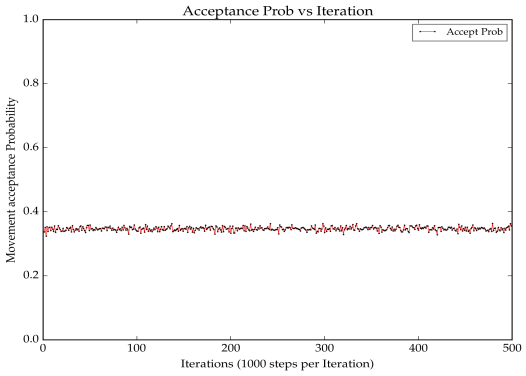
(d) T = 25K, PE vs iteration



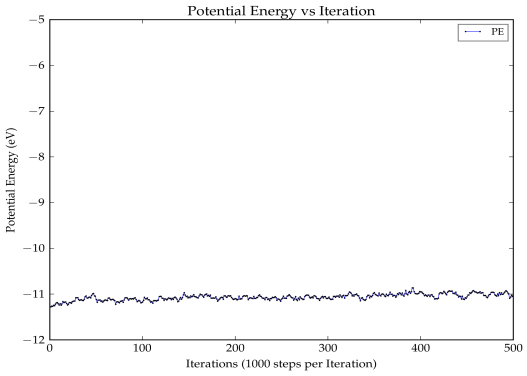
(e) T = 30K, Final Configuration and $P_{hydrostatic}$



(f) T = 30K, RDF

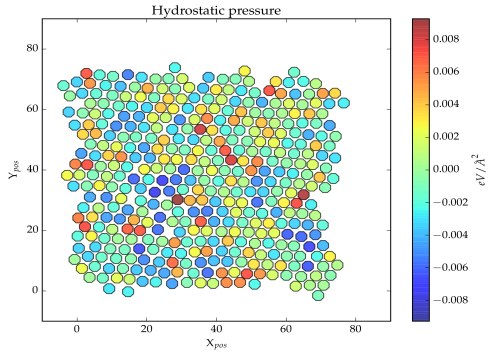


(g) T = 30K, Acceptance probability

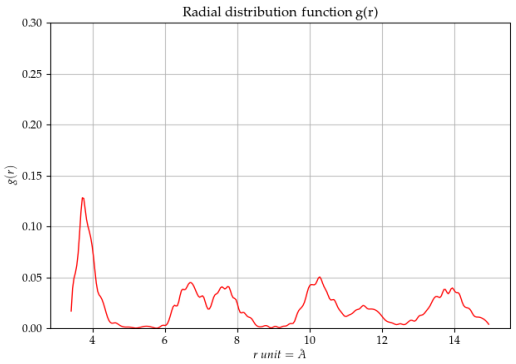


(h) T = 30K, PE vs iteration

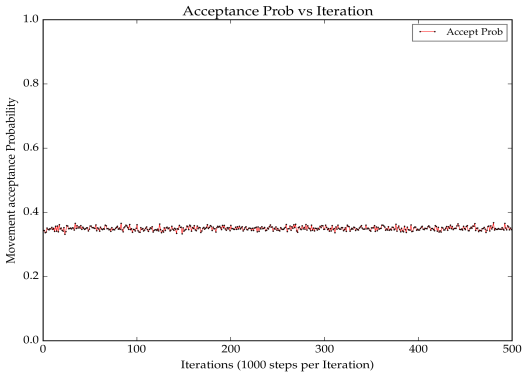
Figure 4: Temperature at 25K and 30K



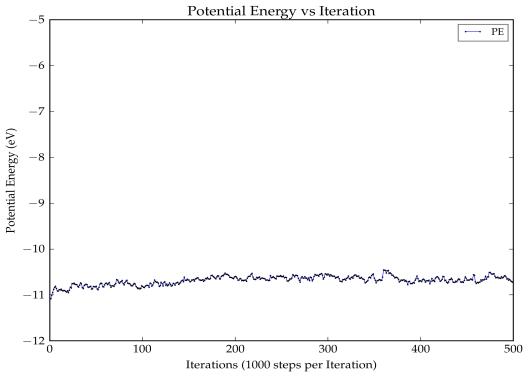
(a) T = 35K, Final Configuration and $P_{hydrostatic}$



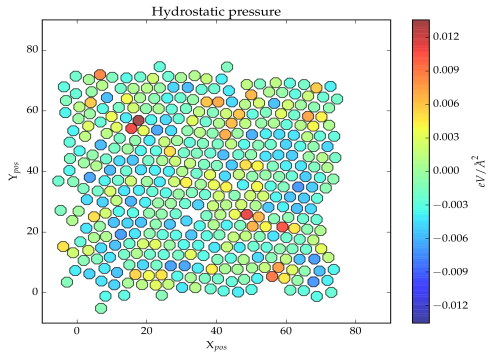
(b) T = 35K, RDF



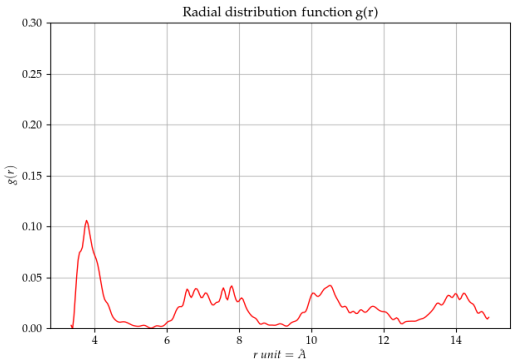
(c) T = 35K, Acceptance probability



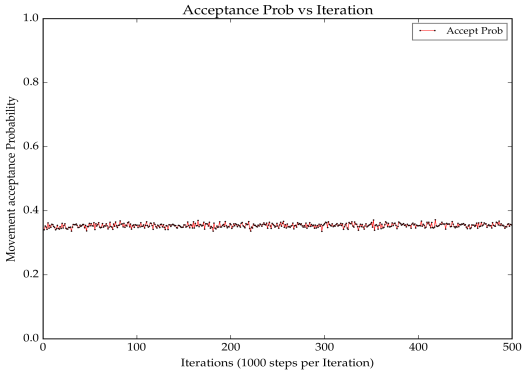
(d) T = 35K, PE vs iteration



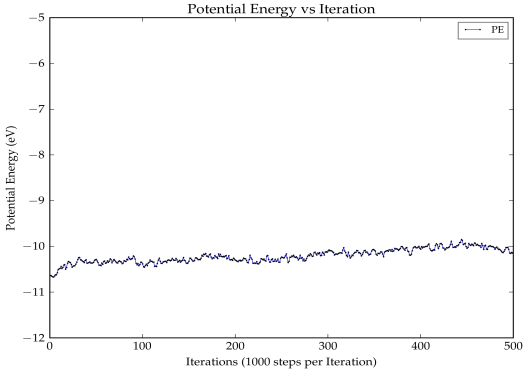
(e) T = 40K, Final Configuration and $P_{hydrostatic}$



(f) T = 40K, RDF

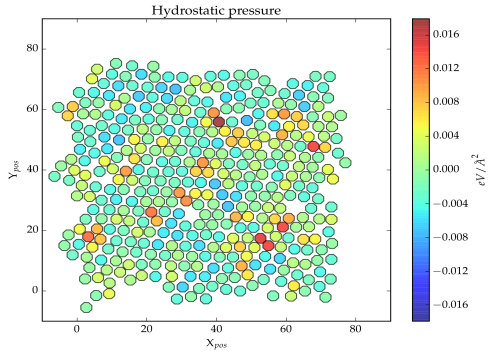


(g) T = 40K, Acceptance probability

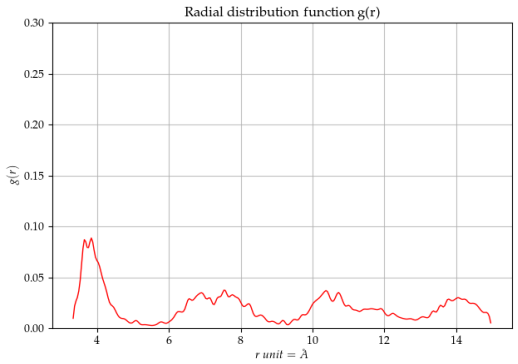


(h) T = 40K, PE vs iteration

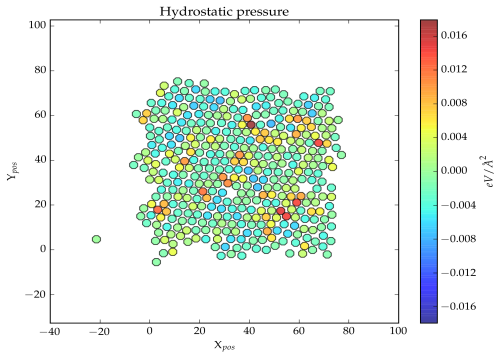
Figure 5: Temperature at 35K and 40K



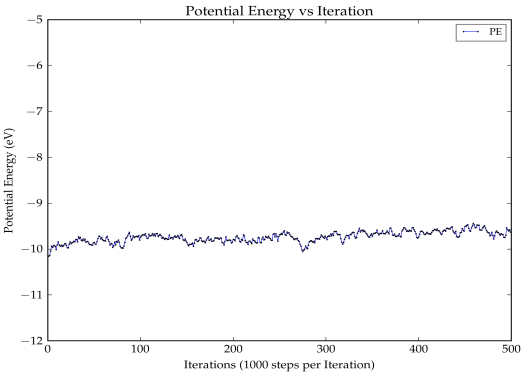
(a) T = 45K, Zoom-in Configuration



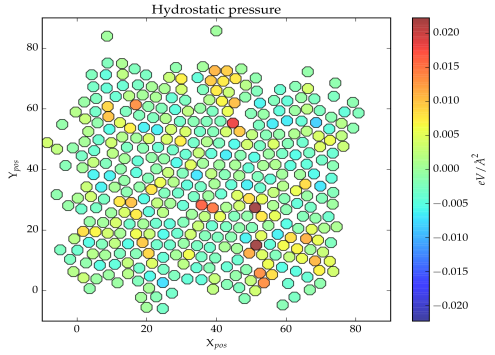
(b) T = 45K, RDF



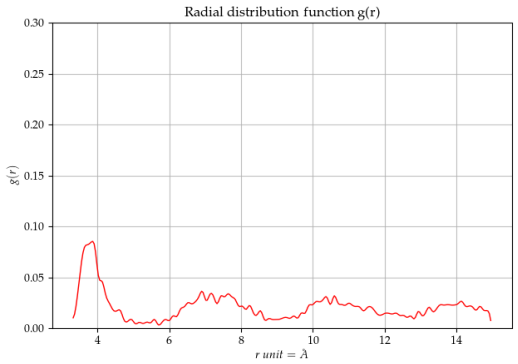
(c) T = 45K, Final Configuration and $P_{hydrostatic}$



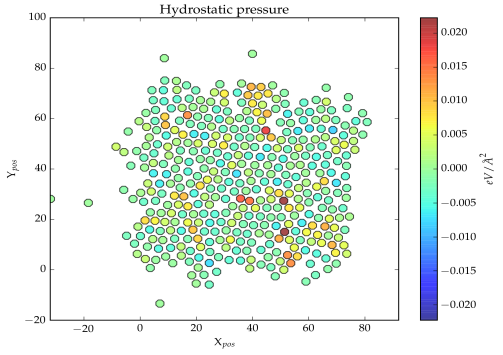
(d) T = 45K, PE vs iteration



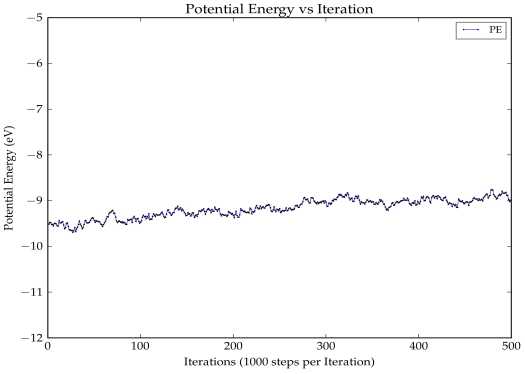
(e) T = 50K, Zoom-in Configuration



(f) T = 50K, RDF

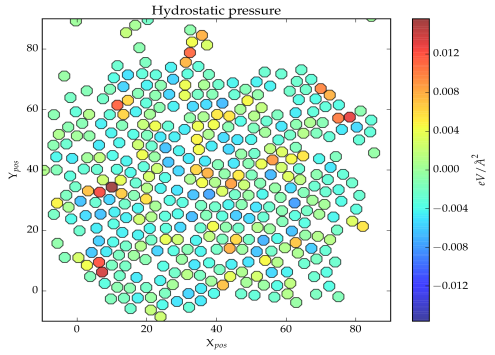


(g) T = 50K, Final Configuration and $P_{hydrostatic}$

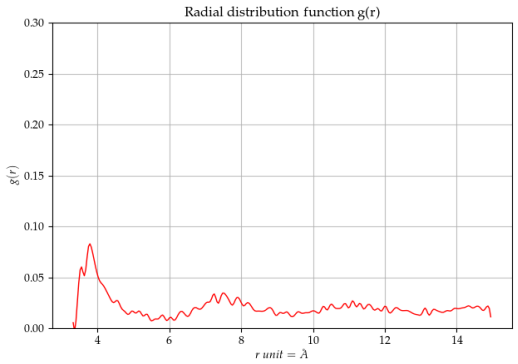


(h) T = 50K, PE vs iteration

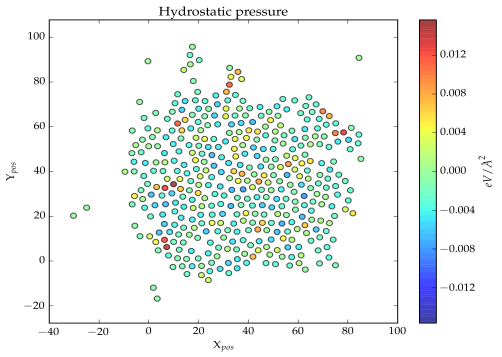
Figure 6: Temperature at 45K and 50K



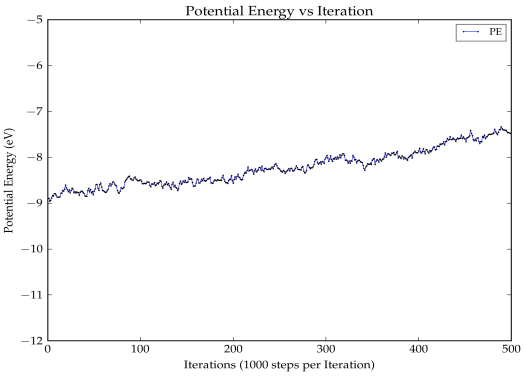
(a) T = 55K, Zoom-in Configuration



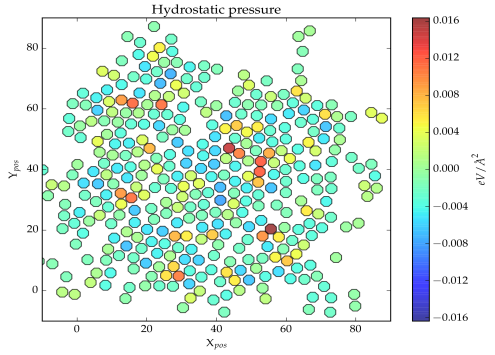
(b) T = 55K, RDF



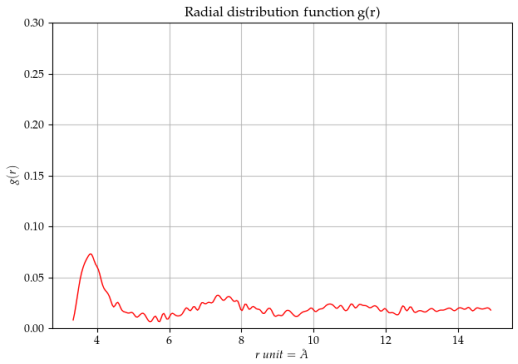
(c) T = 55K, Final Configuration and $P_{hydrostatic}$



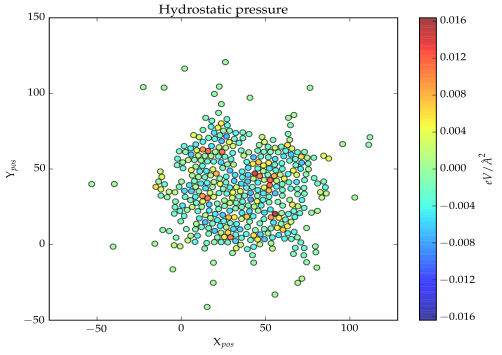
(d) T = 55K, PE vs iteration



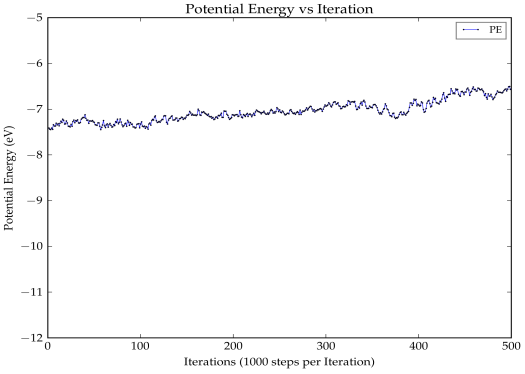
(e) T = 60K, Zoom-in Configuration



(f) T = 60K, RDF

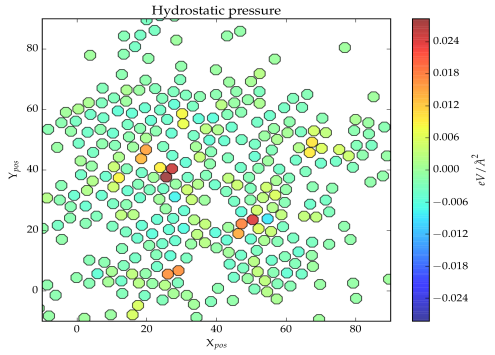


(g) T = 60K, Final Configuration and $P_{hydrostatic}$

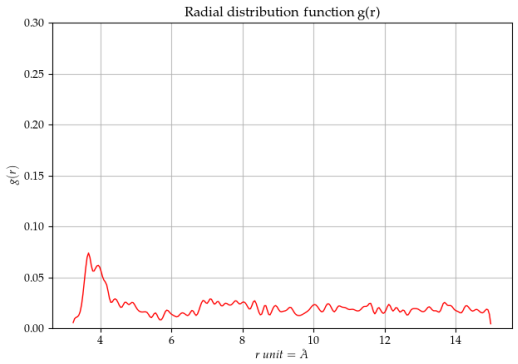


(h) T = 60K, PE vs iteration

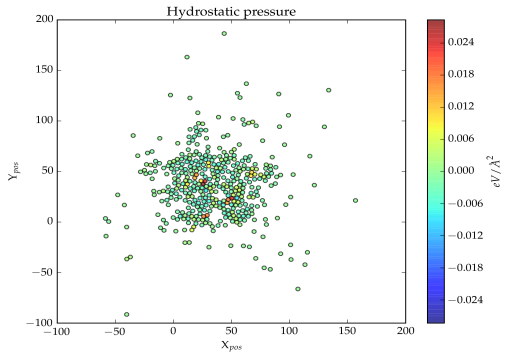
Figure 7: Temperature at 55K and 60K



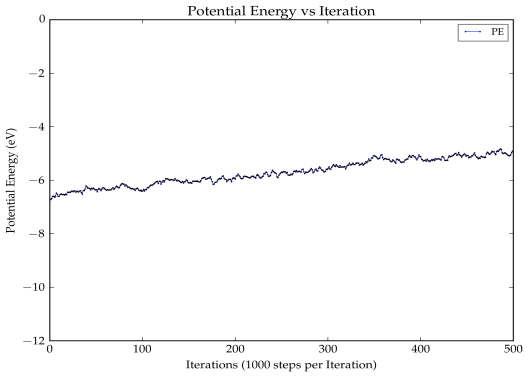
(a) T = 65K, Zoom-in Configuration



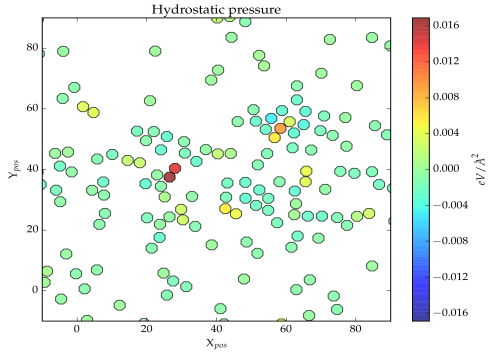
(b) T = 65K, RDF



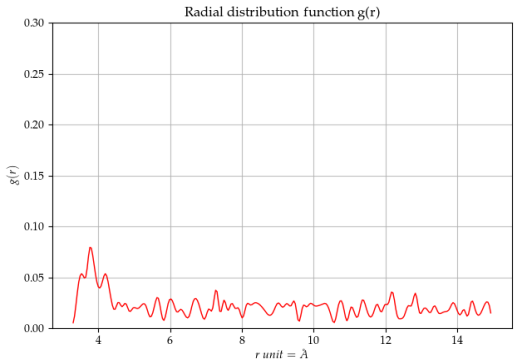
(c) T = 65K, Final Configuration and $P_{hydrostatic}$



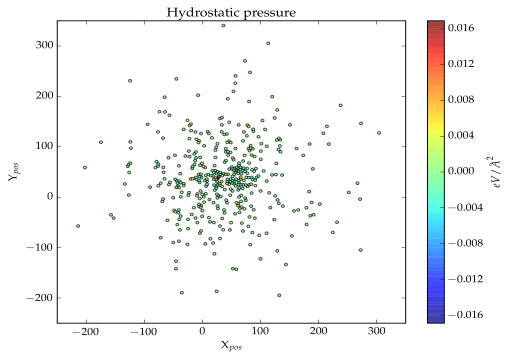
(d) T = 65K, PE vs iteration



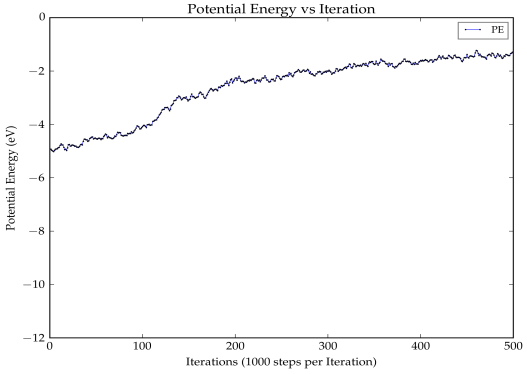
(e) T = 70K, Zoom-in Configuration



(f) T = 70K, RDF

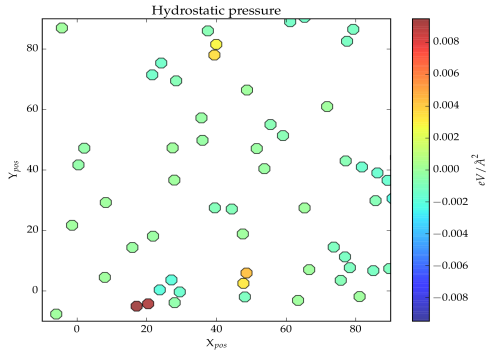


(g) T = 70K, Final Configuration and $P_{hydrostatic}$

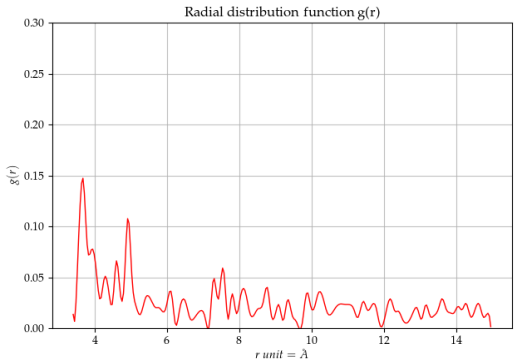


(h) T = 70K, PE vs iteration

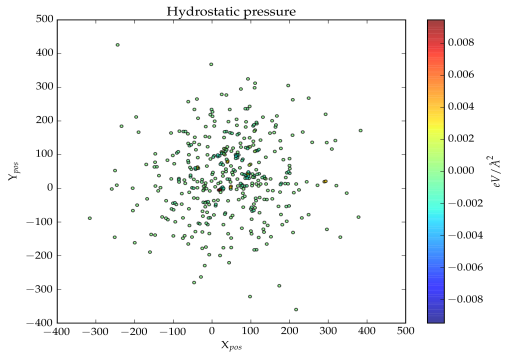
Figure 8: Temperature at 65K and 70K



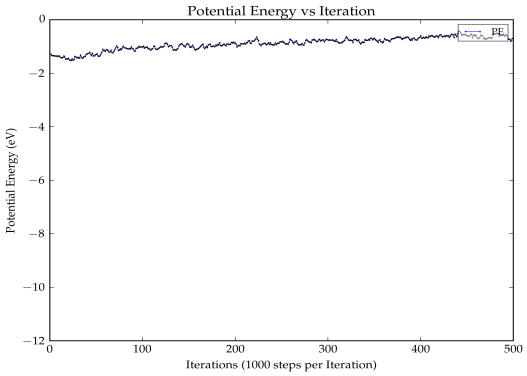
(a) T = 75K, Zoom-in Configuration



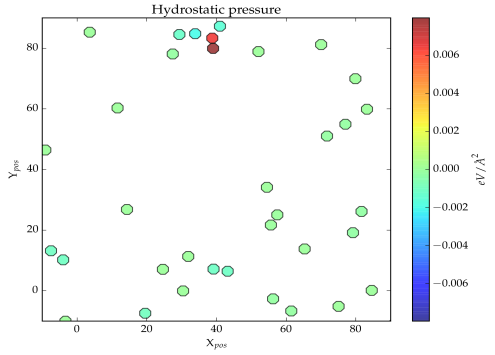
(b) T = 75K, RDF



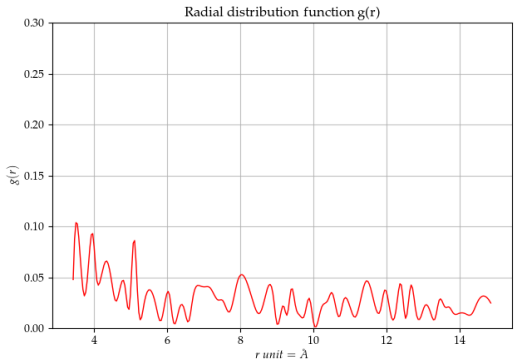
(c) T = 75K, Final Configuration and $P_{hydrostatic}$



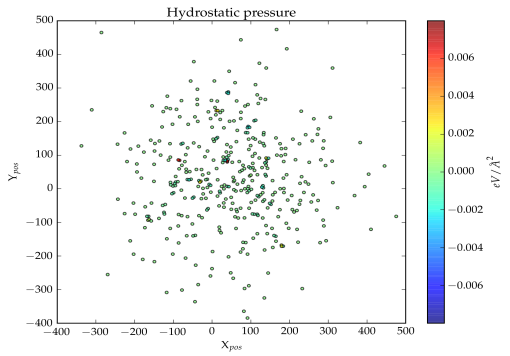
(d) T = 75K, PE vs iteration



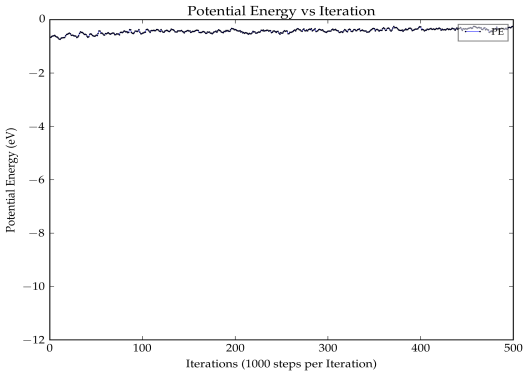
(e) T = 80K, Zoom-in Configuration



(f) T = 80K, RDF

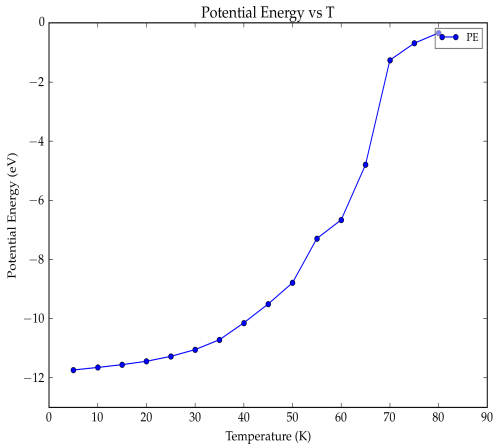


(g) T = 80K, Final Configuration and $P_{hydrostatic}$

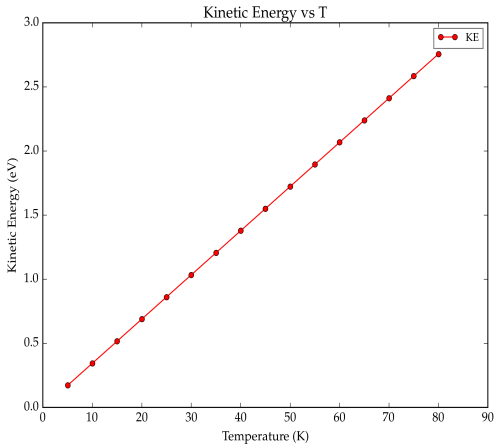


(h) T = 80K, PE vs iteration

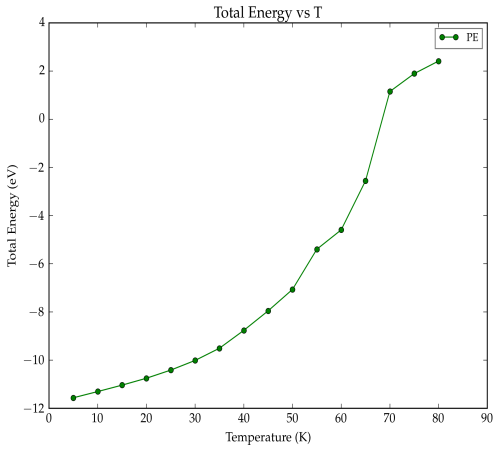
Figure 9: Temperature at 75K and 80K



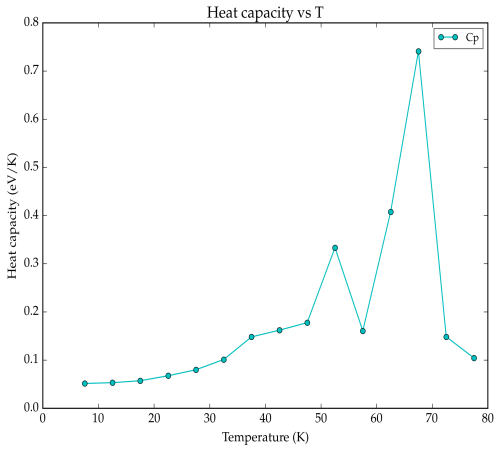
(a) Potential Energy vs Temperature



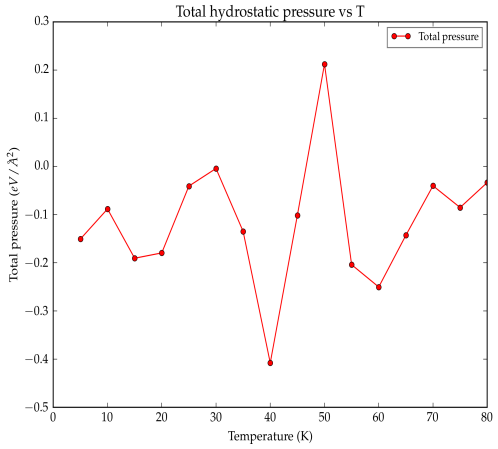
(b) Kinetic Energy vs Temperature



(c) Total Energy vs Temperature



(d) Heat Capacity vs Temperature



(e) Total pressure vs Temperature

Figure 10: Energy,C_p and P_{tot} at different Temperatures

3 Conclusion

1. Same as we got from Molecular dynamics simulations, phase transformation from crystalline to liquid start at around 40K and took place from 40K to 60K.

2. In addition, we also observed another phase transformation from liquid to gas from 65K to 80K according to the heat capacity plot, which we have not seen as clearly as in our Molecular dynamics simulation result.

3. The Heat capacity, which has been calculated by its definition based on the plot of E vs T, shows two peaks from 45K to 60K and from 60K to 75K which agree with our prediction that there are two phase transformations. (crystalline to liquid and liquid to gas)

4. From 5K to 35K, the material is crystalline; From 40K to 60K, phase transformation took place, so the material is a mixture of crystalline and liquid. After 60K, the material is dominated by a liquid phase and from 65K to 80K, a liquid to gas phase transformation has taken place such that the material would be a mixture of liquid and gas.

5. The pressure vs Temperature plot shows that the pressure start with about 0 and then there is a increment of magnitude as the block of atoms goes toward to phase transformation and then finally when the block reaches the final gas state, the pressure goes back to zero, which is expected.

6. Compare this work to our previous Molecular dynamics study: the overall results are quite similar and both simulation shows a clear phase transformation from crystalline to liquid taking place at about 40K to 60K. However there are 3 main differences: **A.** in this work, we have also seen a good evidence (from the heat capacity plot) of phase transformation from liquid to gas at about 70K to 80K which we have not observed in Molecular dynamics. **B.** Monte carlo simulation also takes longer running time than Molecular dynamics that one need 500000 steps for 1 Monte Carlo simulation (takes me 1 hour) and with 30000 steps (take about 10 min) for 1 Molecular dynamics simulation. **C.** Molecular dynamics simulation relies heavily on starting configurations and each of the 16 simulations (from 5K to 80K) actually depend on each other because the whole simulating process is solving the equation of motion thus have physical meanings, and hence we have to run the set of simulations continuously. Nevertheless, for Monte carlo, the only thing that has physical meaning is the final equilibrium state so that before reaching the equilibrium, everything happened in between has no physical meanings and therefore each simulation is basically independent to each other.

4 Appendix: Source code

Firstly, thanks very much for reading the source code. This work is done by a combination of C++ and python where C++ takes care of all the simulation and data collection while python is in charge of all the plotting and data visualization. The implementation of C++ code for the simulation is briefly described in the header file 2dblock.h.

```
// 2dblock.h
// HW5 Monte Carlo Simulation
// 2-D Block
// Author: Yuding Ai
// Date: 2017.04.11

#ifndef BLOCK_H
#define BLOCK_H
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <cmath>
#include <array>
#include <algorithm>
#include <vector>
using namespace std;

/* README
 * This implementation is build on top of my assignment 4
 * As always, thanks!
 */


///-----------------------------------------------------------
//Constants  (following the class note and Spencer's reference code)
///-----------------------------------------------------------
const double epsilon = 0.010323;
const double sigma = 3.405;
const double rMin = 3.822;
const double rTail = 7.0;
const double rCut = 7.5;
const double A = -0.0068102128;
const double B = -0.0055640876;
const double rdfMin = 1.0;
const double rdfMax = 15.0;
const double deltaR = 0.1;
const double neighCut = 7.5;
const double PI = 3.1415926;

const double k_b = 8.617*1E-5; //Boltzman's constant eV*K^-1
const double mass = 39.948*1.66E-27; // the mass of argon atom kg

// const double delta_t = 1E-15; // artifitially chosen the time step for MD
const double delta_t = 1E-14; // artifitially chosen the time step for MD
```

```
47
class block
49 {
    private:
51      int w;          // Width;
        int h;          // Height;
53      int n;          // number of atoms;

55      // each atom is represented by an array
        array<array<double,9>,400> atomlist;
57      // Once the idx is known, all the info about idxth atom is known
        // because we could reach such atom by: block[idx]
59
        //a list of neighbor lists of each atom
61      array<vector<int>,400> neighbor_list;

63  public:
        ///------------constructor-----------------------------------
65      //initialize the block with n atoms and arranged to initial
        //configuration. (the one in assignment 2)
67      //And set the 3 stresses into 0 by default
        ///---------------------------------------------------------
69      block(int x,int y,int N);

71      ///---------------------------------------------------------
        /// getter (subroutines)
73      ///---------------------------------------------------------

75      int getN();

77      // With this operator[], once the idx is known, all the info about
        // idxth atom is known because we could reach such atom by: block[idx]
79      // for example: its position is (block[idx][0],block[idx][1]) and its
        // stresses are (block[idx][2],block[idx][3],block[idx][4])
81      array<double,9> &operator[](int index);


83
        // return the distance between two atoms ----r_{ik}
85      double get_distance(int idx1, int idx2) const;

87      // return the distance in x axis between two atoms ----r_{ik}^{alpha}
        double get_xdistance(int idx1, int idx2) const;
89
        // return the distance in y axis between two atoms ----r_{ik}^{beta}
91      double get_ydistance(int idx1, int idx2) const;

93      //------------get_neighbor_list-----------------------------------
        // take a single argument r, which is neighCut in this assignment
95      // and will update neighbor_list of current configuration.
        // Using Spencer's trick to reduce the computational complicity
97      // ---------------------------------------------------------
        void get_neighbor_list(double r);
99
        //------------get_atomic_stress-----------------------------------
```

```
        //take no argument and compute the atomic stresses of all the
        //atoms and assign it to each atom.
        // ------------------------------------------------------------
        void get_atomic_stress();

        //------------rdf() ------------------------------------------
        //Get the radial distribution and output into a txt file
        // ------------------------------------------------------------
        void rdf(string filename);


        //------------out_config()-------------------------------------
        // output the configuration of atoms for latter plotting
        // and visualization
        // ------------------------------------------------------------
        void out_config(string filename);

        //------------total_penergy()-----------------------------------
        // compute the total potential energy of a configuration by sum over
        // the potential energy of all atoms
        // E_tot = 1/2 sum_{i,j} psi(r_{i,j})
        // ------------------------------------------------------------
        double total_penergy();

        //------------calc_force() & calc_single_force()---------------
        //compute the force of a configuration for each atom by:
        //F_i = -grad_{r_i}*psi(r_{i,j}); where j are neighbors of i
        //calc_force()store the force for each atom into a list and
        //then find and return the maximum force;
        //
        //calc_single_force()
        //compute the net force on a single atom and assign such
        //force onto atomlist[idx][5] and atomlist[idx][6]
        //
        //qcalc_mforce() calculate the maximum force a current atomlist
        //by simply sort the force component
        // ------------------------------------------------------------
        double calc_force();
        void calc_single_force(int idx);
        double qcalc_mforce();


        //------------calc_total_stress_pressure()---------------------
        //compute the total stress sigma_xx, simga_yy and simga_xy and
        //the hydrostatic pressure P. Store the 4 values into an array
        //such that (sum_simga_xx,sum_simga_yy,sum_simga_xy,P)
        //------------------------------------------------------------
        array<double,4> calc_total_stress_pressure();

        //------------SD(double lambda)--------------------------------
        //perform a steepest decent minimization
        //in my case, for each SD, perform 8000 iterations
        // ------------------------------------------------------------
        void SD(double lambda,int it,string filename);
```

```
155
        //===============Methods for HW 4==============================
157
        /*----------- set_config(filename) ----------------------------
159      * manully set the configuration for the block,so including
         * position, velocities, Atomic Force and so on (9 value per atom)
161      */
        void set_config(string filename);
163
        /*-----------update_position(int idx)--------------------------
165      * update the position of each atom using verlet algorithm
         * then assign the velocity into each atom
167      */
        void update_position();
169
        /*-----------update_velosity(int idx)--------------------------
171      * update the velocity of each atom using verlet algorithm
         * then assign the velocity into each atom
173      */
        void update_velosity(array<double,400> F_oldx, array<double,400>
    F_oldy);
175
        /* ---------- Molecular Dynamics MD_mini() : HW part1 -----------
177      * perform a molecular dynamics simulation using verlet algorithm
         * verlet algorithm  to minimize the potential energy
179      */
        void MD_mini(int it);
181

183     /* ---------- Molecular Dynamics MD() : HW part2----------------
         * perform a molecular dynamics simulation using verlet algorithm
185      * verlet algorithm to simulate the temperature dependence
         */
187     void MD(int it,double desire_T,double old_T,string dir);

189     /* ---------- autocorrelation() --------------------------------
         * calculate the autocorrelation function for each Temperature
191      * the configuration and initial velocity is predefined before
         * we do the molecular dynamics as to calcualte the autocorreation
193      * function
         */
195     void autocorrelation(int it,string dir);

197     /* ---------- self-diffusion coefficient() ----------------------
         * calculate the self-diffusion coefficient D
199      */

201     double diffusion(int it);

203     //===============Methods for HW 5==============================

205     /* --------------------- Move() --------------------------------
         * Attempt to move the chosen atom that
207      * x_new = x_old + xi_x*alpha;
```

```
         * y_new = y_old + xi_y*alpha;
         */
        void Move(double alpha, int &sidx,double &oriXpos, double &oriYpos);

        /* ----------------- redoMove() ---------------------------------
         * in the case of rejecting movement,
         * this method take the block back to its old configuration
         */
        void redoMove(int sidx, double oriXpos, double oriYpos);

        /* ---------------- MonteCarlo() -------------------------------
         * the master method for Monte carlo simulations
         */
        void MonteCarlo(long int steps, double T, string dir);

        /* ------------- CollectEqdata() -------------------------------
         * Once the equlibirum state has reached, we run an additional
         * Monte carlo simulation with fewer steps as to collect
         * the average value of potential energy and hydrostatic pressure
         */
        void CollectEqdata(long int steps,double T,string dir);

};

/// useful functions

// compute Lenard Jones potential
double psi(double r);

// compute the derivative of Lenard Jones potential
double dpsi(double r);
#endif /* 2DBLOCK_H */
```

Listing 1: 2dblock.h

```
// 2dblock.cpp
// HW5 Monte Carlo Simulation
// 2-D Block
// Author: Yuding Ai
// Date: 2017.04.11

#include "2dblock.h"

block::block(int x, int y,int N){
    //set with and hight to x,y
    w = x;
    h = y;
    n = N;

  //initialize the block with n atoms and arranged to initial
    //configuration. (the one in assignment 2)
    //And set the 3 stresses into 0 by default
    for (int i = 0; i < n; i++) {
        double xpos,ypos;
```

```
        xpos = (i%w)* rMin;  // default spacing between atoms to be rMin
        ypos = (i/h)* rMin;
        array<double,9> atom;
        atom[0] = xpos;
        atom[1] = ypos;
        atom[2] = atom[3] =atom[4] = atom[5] = atom[6]=0.0;
        atom[7] = 0; //set initial velocity to be 0;
        atom[8] = 0; //set initial velocity to be 0;
        atomlist[i]=atom;
    }

    // set the initial neighbor list
    get_neighbor_list(neighCut);

    // calculate the initial atomic stresses
    get_atomic_stress();

    // calculate the initial Force on each atom
    calc_force();
}

///------------------------------------------------------------
/// getter (subroutines)
///------------------------------------------------------------

int block::getN(){return atomlist.size();}

array<double,9> &block::operator[](int index){return atomlist[index];}


double block::get_distance(int idx1, int idx2) const {
    array <double,2> coor1 = {{atomlist[idx1][0],atomlist[idx1][1]}};
    array <double,2> coor2 = {{atomlist[idx2][0],atomlist[idx2][1]}};
    double dis;
    dis =sqrt(pow(coor1[0]-coor2[0],2) + pow(coor1[1] - coor2[1],2));
    return dis;
}

double block::get_xdistance(int idx1, int idx2) const {
    array <double,2> coor1 = {{atomlist[idx1][0],atomlist[idx1][1]}};
    array <double,2> coor2 = {{atomlist[idx2][0],atomlist[idx2][1]}};
    double dis;
    dis =coor1[0]-coor2[0];
    return dis;
}

double block::get_ydistance(int idx1, int idx2) const {
    array <double,2> coor1 = {{atomlist[idx1][0],atomlist[idx1][1]}};
    array <double,2> coor2 = {{atomlist[idx2][0],atomlist[idx2][1]}};
    double dis;
    dis =coor1[1]-coor2[1];
    return dis;
}

```

```
void block::get_neighbor_list(double r){
    //first remove previous neighbor list:

    for (int i = 0; i < n; i++) {
        neighbor_list[i].clear();
    }
    //then load the new neighbor list
    // Thanks to Spencer's nice reference code for assignment 2, here
    // we apply the same nice trick to get rid of double counting and
    // cut computational complexity by half

    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            // for this assignment, r = neighCut
            if(get_distance(i,j) <=r){
                neighbor_list[i].push_back(j); // i's new neighbor is j
                neighbor_list[j].push_back(i); // j's new neighbor is i
            }
        }
    }
}

void block::get_atomic_stress(){
    const double omega = pow(rMin,2)*(w-1)*(h-1)/n;

    for(int i = 0; i<n; i++){
        //first remove the previous atomic stress and set it to 0
        atomlist[i][2] = 0;
        atomlist[i][3] = 0;
        atomlist[i][4] = 0;

        for (unsigned int  j = 0; j < neighbor_list[i].size();j++){
            //re calculate the atomic stress
            double r = get_distance(i,neighbor_list[i][j]);
            double rx = get_xdistance(i,neighbor_list[i][j]);
            double ry= get_ydistance(i,neighbor_list[i][j]);
            //sigma_xx
            atomlist[i][2] += dpsi(r)*rx*rx/(r*omega);

            //sigma_yy
            atomlist[i][3] += dpsi(r)*ry*ry/(r*omega);

            //sigma_xy
            atomlist[i][4] += dpsi(r)*rx*ry/(r*omega);
        }
    }
}

void block::rdf(string filename){
    stringstream st;
    const double omega = pow(rMin,2)*(w-1)*(h-1)/n;
    //const double omega = pow(rMin,2)*(800-1)*(800-1)/n;
    vector<double> dislist;
    vector<double> Rlist;
```

```
    vector<double> glist;
129 get_neighbor_list(rdfMax);

131 for (int i = 0; i < 400; i++) {
        for (unsigned int j = 0; j < neighbor_list[i].size(); j++) {
133         dislist.push_back(get_distance(i,neighbor_list[i][j]));
        }
135 }

137 //dislist is sorted in increment order
    sort(dislist.begin(),dislist.end());
139
    double d = dislist[0];
141
    // count each distance
143 int c = 0;
    for (unsigned int i = 0; i < dislist.size(); i++) {
145     if(abs(dislist[i] - d)<= deltaR){
            c++;
147     }
        else{
149         //rlist is a list of possible distance of all neighbors
            //sorted in increment order
151         Rlist.push_back(d);
            //glist is a list of occurrence/frequency of ith r for now
153         glist.push_back(1.0*c/dislist.size()); //divide dislist to
    normalize it
            c = 0;
155         d = dislist[i];
        }
157
        if(i == dislist.size()-1){
159         //rlist is a list of possible distance of all neighbors
            //sorted in increment order
161         Rlist.push_back(d);
            //glist is a list of occurrence/frequency of ith r for now
163         glist.push_back(1.0*c/dislist.size()); //divide dislist.size() to
    normalize it
            c = 0;
165         d = dislist[i];
        }
167 }

169 // last calculate g(r) and update the glist
    for (unsigned int i = 0; i < glist.size(); i++) {
171     // calculate and update glist into g(r) now,
        // namely the radial distribution function
173     glist[i] = glist[i]*omega/(2.0*PI*deltaR*Rlist[i]);
        st<< Rlist[i]<<" "<<glist[i]<<"\n";
175 }

177 // record rdf into a txt file for latter plotting
    ofstream myfile(filename);
179 string data = st.str();
```

```
    myfile<< data;
    myfile.close();

    //last reset the neighbor list to r = neighCut
    //since we modified the r to be rdfMax every time when
    //we calculate the rdf
    get_neighbor_list(neighCut);
}


void block::out_config(string filename){

    stringstream st;
    // record xpos, ypos and stresses of each atom into a txt file
    // file for latter plotting
    for (int i = 0; i < n; i++) {
        st << atomlist[i][0]<<" "<<atomlist[i][1]<<" "<<atomlist[i][2]<<" ";
        // st << atomlist[i][3]<<" "<<atomlist[i][4]<<endl;
        st << atomlist[i][3]<<" "<<atomlist[i][4]<<" "<<atomlist[i][5]<<" ";
        st << atomlist[i][6]<<" "<<atomlist[i][7]<<" "<<atomlist[i][8]<<endl;
    }

    ofstream myfile(filename);
    string data = st.str();
    myfile<< data;
    myfile.close();

}
double block::total_penergy(){
    // this total_penergy is the total potential energy
    double E = 0;
    for (int i = 0; i < n; i++) {
        for (unsigned int j = 0; j < neighbor_list[i].size(); j++) {
            E += psi(get_distance(i,neighbor_list[i][j]));
        }
    }
    // Since E_tot = 1/2 sum_{i,j} psi(r_{i,j})
    E = E/2.0;
    return E;
}

double block::calc_force(){

    double F = 0;
    double curFx = 0;
    double curFy = 0;
    double curF = 0;
    // calc Force on each atom and find the F_max
    for (int i = 0; i < n; i++) {
        curFx = 0;
        curFy = 0;
        for(unsigned int j = 0; j<neighbor_list[i].size();j++){
            double r_ij = get_distance(i,neighbor_list[i][j]);
            double r_ij_x = get_xdistance(i,neighbor_list[i][j]);
```

```
            double r_ij_y = get_ydistance(i,neighbor_list[i][j]);
            curFx+= -dpsi(r_ij)*(r_ij_x/r_ij);
            curFy+= -dpsi(r_ij)*(r_ij_y/r_ij);
        }
        // update the atomlist
        atomlist[i][5]=curFx;
        atomlist[i][6]=curFy;
        curF = sqrt(pow(curFx,2) + pow(curFy,2));
        if(F<curF){
            F = curF;
        }
    }
    return F;
}

void block::calc_single_force(int idx){

    double curFx = 0;
    double curFy = 0;
    // calc Force on each atom and find the F_max
    for(unsigned int j = 0; j<neighbor_list[idx].size();j++){
        double r_ij = get_distance(idx,neighbor_list[idx][j]);
        double r_ij_x = get_xdistance(idx,neighbor_list[idx][j]);
        double r_ij_y = get_ydistance(idx,neighbor_list[idx][j]);
        curFx+= -dpsi(r_ij)*(r_ij_x/r_ij);
        curFy+= -dpsi(r_ij)*(r_ij_y/r_ij);
    }
    // update the atomlist
    atomlist[idx][5]=curFx;
    atomlist[idx][6]=curFy;
}

double block::qcalc_mforce(){
    double maxF = 0;
    double curFx=0;
    double curFy=0;
    double curF=0;
    for (int i = 0; i < n; ++i) {
        curFx=atomlist[i][5];
        curFy=atomlist[i][6];
        curF = sqrt(pow(curFx,2) + pow(curFy,2));
        if(maxF<curF){
            maxF = curF;
        }
    }
    return maxF;
}

array<double,4> block::calc_total_stress_pressure(){
    double sxx,syy,sxy,P;
    // P: total hydrostatic pressure:
    // for each atom i: p_i = -1/2*sum(sxx_i +syy_i)
    // P = sum{i}(p_i)
    sxx = syy = sxy = P = 0;
```

```
    for (int i = 0; i < n; ++i) {
        sxx += atomlist[i][2];
        syy += atomlist[i][3];
        sxy += atomlist[i][4];
        P += -1.0/2.0*(atomlist[i][2] + atomlist[i][3]);
    }

    array<double,4> sum_stress= {{sxx,syy,sxy,P}};
    return sum_stress;
}

//============================== Molecular Statics
    ==============================
void block::SD(double lambda,int it,string filename){
    stringstream st;
    int j = 0;
    int k = 0;
    while(j<it){
        for (int i = 0; i < n; i++) {
            // get the force on each atom
            calc_single_force(i);
            // Move each atom following the direction
            // of it's force
            atomlist[i][0] += lambda*atomlist[i][5];
            atomlist[i][1] += lambda*atomlist[i][6];
        }

        if(it>=1000){
            if(k==it/1000){
                //first update the neighborlist
                get_neighbor_list(neighCut);
                //update the stresses
                get_atomic_stress();
                //calc hydrostatic pressure
                array<double,4> sp = calc_total_stress_pressure();
                //calc total potential energy
                double E = total_penergy();
                //calc maximum force
                double F = qcalc_mforce();
                //record the state/configuration data
                st<<j<<" "<<E<<" "<<sp[0]<<" "<<sp[1]<<" ";
                st<<sp[2]<<" "<<sp[3]<<" "<<F<<endl;
                k = 0;
            }
        }
        else{
            //first update the neighborlist
            get_neighbor_list(neighCut);
            //update the stresses
            get_atomic_stress();
            //calc hydrostatic pressure
            array<double,4> sp = calc_total_stress_pressure();
            //calc total energy
            double E = total_penergy();
```

```
341             //calc maximum force
                double F = qcalc_mforce();
343             //record the state/configuration data
                st<<j<<" "<<E<<" "<<sp[0]<<" "<<sp[1]<<" ";
345             st<<sp[2]<<" "<<sp[3]<<" "<<F<<endl;
            }
347
            j++;
349         k++;
        }
351     ofstream myfile(filename);
        string data = st.str();
353     myfile<< data;
        myfile.close();
355 }

357 //=================== Molecular dynamics ============================
    //--------------------HW 4 starts from here  ----------------------
359 //====================================================================
    void block::set_config(string filename){
361     string line;
        ifstream myfile(filename);
363     if(myfile.is_open()){
            int i = 0;
365         while (std::getline(myfile,line)){
                stringstream linestream(line);
367             string data;
                std::getline(linestream,data,' ');
369             atomlist[i][0] = std::stod(data);
                linestream >>atomlist[i][1]>>atomlist[i][2]>>
371                 atomlist[i][3]>>atomlist[i][4]
                    >>atomlist[i][5]>>atomlist[i][6]
373                 >>atomlist[i][7]>>atomlist[i][8];
                i++;
375         }
        }
377 }
    void block::update_position(){
379     /* According to Verlet algorithm
         * r_i((j+1)*deltat) = r_i(j*deltat) + v_i(j*deltat)*deltat +
381      * (deltat)^2/2m_i *F_i(j*deltat)
         */
383
        //Recall in my atom's data structure,
385     //atom[0]  --- xposition
        //atom[1]  --- yposition
387     //atom[5]  --- F_x
        //atom[6]  --- F_y
389     //atom[7]  --- v_x
        //atom[8]  --- v_y
391     //And when we calc r, F and v are all corresponding to j*delte_t
        for(unsigned int idx = 0; idx< atomlist.size();idx++){
393         //x_position
            atomlist[idx][0] = atomlist[idx][0] + atomlist[idx][7]*delta_t
```

```
395             + delta_t*delta_t/(2*mass)* atomlist[idx][5];

397         //y_position
            atomlist[idx][1] = atomlist[idx][1] + atomlist[idx][8]*delta_t
399             + delta_t*delta_t/(2*mass)* atomlist[idx][6];
    }
401 }

403 void block::update_velosity(array<double,400> F_oldx, array<double,400> F_oldy
    ){
    /* According to Verlet algorithm
405  * v_i((j+1)*deltat) = v_i(j*deltat) + deltat/2m_i*(F_i((j+1)*deltat)
     * + F_i(j*deltat) )
407  *
     * To have F_i((j+1)*delta_t), we should first store the old Force
409  * F_i(j*delta_t) then update the force corresponding to (j+1)*delta_t
     * configuration
411  */
    for(unsigned int idx = 0; idx< atomlist.size();idx++){
413     //v_x
        atomlist[idx][7] = atomlist[idx][7] + delta_t/(2*mass) *(atomlist[idx
    ][5]+ F_oldx[idx]);
415
        //v_y
417     atomlist[idx][8] = atomlist[idx][8] + delta_t/(2*mass) *(atomlist[idx
    ][6]+ F_oldy[idx]);
    }
419 }

421 void block::MD_mini(int it){
    stringstream st;
423 stringstream st_temp_stress;
    array<double,4> totstress; //store the total stress and pressure
425 array<double,10> pressurearray; //store the pressure of the 10 MD
    simulations
    array<double,10> temperaturearray; //store the temperature of the 10 MD
    simulations
427 array<double,400> F_oldx;
    array<double,400> F_oldy;
429
    int counter = 0;
431 //I have tried many runs on this model and
    //according to my experiments:
433 //when delta_t = 1E-14:
    //A. At least 30000 iterations is needed for the first MD
435 //simulation to reach the equalibirum. For the next few MD simulations
    //, fewer and fewer steps is needed to reach its equalibirum. so
437 //here we choose 50000 iterations for each MD simulation.
    //and after 30000 iterations, we start to collect date as to calculate
439 //the equalibirum average.
    //
441 //B. I found that 10 times of sucessive MD simulation with 50000
    //iterations for each MD simulation would be sufficient enough
443 //to reach the minimun potential energy.(PE_min = -12.3343 eV) which is
```

```
    //even better than the result from my previous Molecular static method
445 //(-11.7813 eV)
    //
447 //Therefore, here we do 10 successive MD simulations to minimize
    //the potential energy.
449
    for (int n = 0; n<10;n++){
451     //For each MD, before looping, we first set vx and vy to be k_fac*v;
        for(unsigned int j = 0; j< atomlist.size();j++){
453         atomlist[j][7] = 0;
            atomlist[j][8] = 0;
455     }
        double KE= 0;
457     double KEsum = 0;
        double Psum=0;
459
        //then get the initial Force and potential energy
461     double PE = total_penergy();
        calc_force();
463
        int i = 0;
465     while(i < it){
            for(unsigned int j = 0; j< atomlist.size();j++){
467             F_oldx[j] = atomlist[j][5];
                F_oldy[j] = atomlist[j][6];
469         }
            //for eacn iteration, first get the new position
471         update_position();
            //then once atoms are moved, we update the neighbor_list
473         get_neighbor_list(neighCut);
            // calc the new force
475         calc_force();
            //then get the new velocity(the new force is been updated during
477         update_velosity(F_oldx, F_oldy);

479         //calc the PE and KE corresponding to the new configuration
            PE = total_penergy();
481         KE = 0; //recalc the KE
            for (unsigned int k = 0; k<atomlist.size();k++){
483             KE += 0.5* mass*(atomlist[k][7]*atomlist[k][7] +
                        atomlist[k][8]*atomlist[k][8]);
485         }
            //store the infomation into stringtream
487         st<< KE <<" "<<PE<<" "<<counter<<endl;
            cout<< KE <<" "<<PE<<" "<<counter<<endl;
489         i++;
            counter++;
491
            //According to my result, for delta_t = 1E-14, equalibirum would
    reached
493         //after 30000 iterations, so from there, we start to record the KE
     value
            //as to find the average <KE>, then to obtain temperature by N*k_b
    *T = <KE>
```

```
            //FYI, My program takes: 4213.73 sec to compele 10 sets of MD simulation
            //with 50000 itetarions each. so the computation is pretty slow

            if(i>30000){
                KEsum +=KE;
                totstress = calc_total_stress_pressure();
                Psum += totstress[3];
            }
        }

        KEsum /=(it - 30000); //get the <KE>
        Psum /=(it - 30000); //get the <P>
        double temp = KEsum/(400*k_b);
        get_atomic_stress();
        totstress = calc_total_stress_pressure();
        pressurearray[n] = totstress[3];
        temperaturearray[n] = temp;

        st_temp_stress<<temp <<" "<<totstress[0]<<" "<<totstress[1]<<" "<<totstress[2]
            <<" "<<totstress[3]<<" "<<Psum<<endl;
    }

    ofstream myfile1("MD_temp_stress.txt");
    string data1 = st_temp_stress.str();
    myfile1<< data1;
    myfile1.close();

    ofstream myfile("MD_Energy.txt");
    string data = st.str();
    myfile<< data;
    myfile.close();
}


void block::MD(int it,double desire_T,double old_T,string dir){
    // This program is basically the same as MD_mini using the same algorithm
    // but this time is to simulate the temperature dependence MD
    stringstream st;
    stringstream st_temp_stress;
    array<double,4> totstress; //store the total stress and pressure
    array<double,400> F_oldx;
    array<double,400> F_oldy;

    int counter = 0;

    //before starts the MD simulation, we first calculate k_factor based on the
    //choice of desire_T and a preknown old_T which is obtained from previous simulation;
    //the very initial state, set old_T to be 1 and desire_T =  0
    //
    //as K = (2Nk_b*T_desire/sum(m_i*<vi^2>))^0.5 = (T_desire/T_old)^0.5
```

```
545
    double k_fac = pow(desire_T/old_T,0.5); //set initial k_fac
547 // for (int n = 0; n<20;n++){
    for (int n = 0; n<30;n++){
549     //For each MD, before looping, we first scale the velocity
        for(unsigned int j = 0; j< atomlist.size();j++){
551         atomlist[j][7] = atomlist[j][7]*k_fac;
            atomlist[j][8] = atomlist[j][8]*k_fac;
553     }
        double KE= 0;
555     double velocity = 0;
        double KEsum_tempdenpendent = 0;
557     double hydroP=0;
        double Tdependent_temp = 0;
559
        //then get the initial Force and potential energy
561     double PE = total_penergy();
        calc_force();
563
        int i = 0;
565     while(i < it){
            //get the old force
567         for(unsigned int j = 0; j< atomlist.size();j++){
                F_oldx[j] = atomlist[j][5];
569             F_oldy[j] = atomlist[j][6];
            }
571         //for eacn iteration, first get the new position
            update_position();
573         //then once atoms are moved, we update the neighbor_list
            get_neighbor_list(neighCut);
575         // calc the new force
            calc_force();
577         //then get the new velocity(the new force is been updated during
            update_velosity(F_oldx, F_oldy);
579
            //calc the PE and KE corresponding to the new configuration
581         PE = total_penergy();
            KE = 0; //recalc the KE
583         for (unsigned int k = 0; k<atomlist.size();k++){
                KE += 0.5* mass*(atomlist[k][7]*atomlist[k][7] +
585                     atomlist[k][8]*atomlist[k][8]);
                velocity += pow(atomlist[k][7]*atomlist[k][7]+
587                     atomlist[k][8]*atomlist[k][8],0.5);

589         }

591         get_atomic_stress();
            totstress = calc_total_stress_pressure();
593         velocity /=atomlist.size();
            cout<< KE <<" "<<PE<<" "<<" "<<velocity<<" "<<counter<<" "<<
totstress[3]<<endl;
595         st<< KE <<" "<<PE<<" "<<" "<<velocity<<" "<<counter<<" "<<
totstress[3]<<endl;
            i++;
```

```
            counter++;
            KEsum_tempdenpendent +=KE;
            hydroP +=totstress[3];

        }

        //record the final config of each MD simulation
        KEsum_tempdenpendent /= (it); //get <KE>
        hydroP /= (it); //get <P>
        // autocorr /=(it-1000); //get <v(t)v(0)>
        Tdependent_temp = KEsum_tempdenpendent/(400*k_b);
        get_atomic_stress();
        totstress = calc_total_stress_pressure();

        cout<<k_fac<<" "<<Tdependent_temp<<" "<<i<<endl;
        st_temp_stress<<Tdependent_temp <<" "<<totstress[0]<<" "<<totstress[1]
            <<" "<<totstress[2]<<" "<<totstress[3]<<" "<<hydroP<<endl;

        //update the k_fac
        k_fac = pow(desire_T/Tdependent_temp,0.5);
    }

    string filename1 = dir +"/part2_MD_temp_stress.txt";
    ofstream myfile1(filename1);
    string data1 = st_temp_stress.str();
    myfile1<< data1;
    myfile1.close();

    string filename = dir +"/part2_MD_Energy.txt";
    ofstream myfile(filename);
    string data = st.str();
    myfile<< data;
    myfile.close();
}

void block::autocorrelation(int it,string dir){
    stringstream st;
    stringstream st_temp_stress;
    array<double,400> F_oldx;
    array<double,400> F_oldy;
    vector<array<double,400> > v_all;

    int counter = 0;

    double KE= 0;
    double KEsum_tempdenpendent = 0;

    //then get the initial Force and potential energy
    double PE = total_penergy();
    calc_force();

    int i = 0;
    counter = 0;
```

```
651     double velocity = 0;
        while(i < it){
653         velocity = 0;
            //get the old force
655         for(unsigned int j = 0; j< atomlist.size();j++){
                F_oldx[j] = atomlist[j][5];
657             F_oldy[j] = atomlist[j][6];
            }
659         //for eacn iteration, first get the new position
            update_position();
661         //then once atoms are moved, we update the neighbor_list
            get_neighbor_list(neighCut);
663         // calc the new force
            calc_force();
665         //then get the new velocity(the new force is been updated during
            update_velosity(F_oldx, F_oldy);
667
            //calc the PE and KE corresponding to the new configuration
669         PE = total_penergy();
            KE = 0; //recalc the KE
671         array<double,400> v_run;
            for (unsigned int k = 0; k<atomlist.size();k++){
673             KE += 0.5* mass*(atomlist[k][7]*atomlist[k][7] +
                        atomlist[k][8]*atomlist[k][8]);
675
                velocity = pow(atomlist[k][7]*atomlist[k][7]+
677                     atomlist[k][8]*atomlist[k][8],0.5);
                v_run[k] = velocity;
679         }
            v_all.push_back(v_run);
681         cout << counter<<"velocity "<<KE <<" "<<velocity<<endl;

683         i++;
            counter++;
685         KEsum_tempdenpendent +=KE;
        }
687
        double autoco = 0;
689     double vo = 0;
        double vt = 0;
691     for (int k = 0; k<5000;k++){
            autoco = 0;
693         for(int i = 0; i<5000;i++ ){
                for(int j = 0; j<400;j++){
695                 vo = v_all[i][j];
                    vt = v_all[i+k][j];
697                 autoco = autoco + vo*vt;
                }
699         }
            autoco = autoco/(400*5000);
701         st<<autoco<<" "<<k<<endl;
        }
703
        string filename =dir + "/auto.txt";
```

```
705     ofstream myfile(filename);
        string data = st.str();
707     myfile<< data;
        myfile.close();
709 }

711 double block::diffusion(int it){
        stringstream st;
713     stringstream st_temp_stress;
        array<double,400> F_oldx;
715     array<double,400> F_oldy;
        vector<array<double,400> > r_x;
717     vector<array<double,400> > r_y;

719     int counter = 0;

721     double KE= 0;
        double KEsum_tempdenpendent = 0;
723
        //then get the initial Force and potential energy
725     double PE = total_penergy();
        calc_force();
727
        int i = 0;
729     counter = 0;

731     double velocity = 0;
        while(i < it){
733         velocity = 0;
            //get the old force
735         for(unsigned int j = 0; j< atomlist.size();j++){
                F_oldx[j] = atomlist[j][5];
737             F_oldy[j] = atomlist[j][6];
            }
739         //for eacn iteration, first get the new position
            update_position();
741         //then once atoms are moved, we update the neighbor_list
            get_neighbor_list(neighCut);
743         // calc the new force
            calc_force();
745         //then get the new velocity(the new force is been updated during
            update_velosity(F_oldx, F_oldy);
747
            //calc the PE and KE corresponding to the new configuration
749         PE = total_penergy();
            KE = 0; //recalc the KE
751         array<double,400> r_runx;
            array<double,400> r_runy;
753         for (unsigned int k = 0; k<atomlist.size();k++){
                KE += 0.5* mass*(atomlist[k][7]*atomlist[k][7] +
755                     atomlist[k][8]*atomlist[k][8]);

757             r_runx[k] = atomlist[k][0];
                r_runy[k] = atomlist[k][1];
```

```
759         }
            r_x.push_back(r_runx);
761         r_y.push_back(r_runy);
            cout << counter<<"velocity "<<KE <<" "<<velocity<<endl;
763
            i++;
765         counter++;
            KEsum_tempdenpendent +=KE;
767     }

769     double rox = 0;
        double roy = 0;
771     double rtx = 0;
        double rty = 0;
773     double D= 0;
        for (int k = 0; k<5000;k++){
775         D = 0;
            for(int i = 0; i<5000;i++ ){
777             for(int j = 0; j<400;j++){
                    rox = r_x[i][j];
779                 roy = r_y[i][j];
                    rtx = r_x[i+k][j];
781                 rty = r_y[i+k][j];
                    D += (rtx-rox)*(rtx-rox)+(rty-roy)*(rty-roy);
783             }
            }
785         D = D/(400*5000);
        }
787     D /= (6*5000);
        double lnD=0;
789     lnD = log(D);
        return lnD;
791 }

793 //======================== Monte Carlo Simulation  ========================
    //-------------------------HW 5 starts from here  -------------------------
795 //=========================================================================
    void block::Move(double alpha, int &sidx,double &oriXpos, double &oriYpos){
797     //first randomly pick an atom from the 400 atom block:
        int ridx = rand()%int(400); //ridx is in range 0 to 399;
799     sidx = ridx;

801     //next, generate two random numbers zetax and zetay
        double zetax = -1.0 + 2.0*(double)rand()/ RAND_MAX;
803     double zetay = -1.0 + 2.0*(double)rand()/ RAND_MAX;

805     oriXpos = atomlist[ridx][0];
        oriYpos = atomlist[ridx][1];
807
        //now move that atom by: x' = x + alpha*zetax;   y' = y + alpha*zetay;
809     atomlist[ridx][0] += alpha*zetax;
        atomlist[ridx][1] += alpha*zetay;
811     //cout<<"xmove = "<<alpha*zetax <<"ymove = "<<alpha*zetay<<endl;
    }
```

```
813
void block::redoMove(int sidx, double oriXpos, double oriYpos){
815     atomlist[sidx][0] = oriXpos;
        atomlist[sidx][1] = oriYpos;
817 }

819 void block::MonteCarlo(long int steps,double T,string dir){
        stringstream st;
821     long int it = 0;
        // double KE = 400*k_b*T;
823     double PE = 0;
        double PEave = 0;
825     int counter = 0;
        double alpha = 1;
827     double sumalpha = 0;
        double probacceptave = 0;
829     while(it<steps){
            PE = total_penergy();
831         double oriXpos,oriYpos,newPE,probaccept;
            double prob;
833         int sidx;
            // move the atom
835         Move(alpha,sidx,oriXpos,oriYpos);

837         // once we change the config, we update neighborlist and the PE
            get_neighbor_list(neighCut);
839         newPE = total_penergy();

841         // check the acceptance prob;
            if(newPE>PE){
843             probaccept = exp(-(newPE-PE)/(k_b*T));
                prob = (double)rand()/ RAND_MAX;
845
                if(prob > probaccept){
847                 // do not accept if prob > probaccept
                    redoMove(sidx,oriXpos,oriYpos);
849             }
                else{
851             // else accept move
                    PE = newPE;
853             }
            }
855         else{
                probaccept = 1;
857             // accept the move
                PE = newPE;
859         }

861         sumalpha += alpha;
            PEave += PE;
863         probacceptave += probaccept;

865         if(counter > steps/500){
                sumalpha /=(steps/500);
```

```
867             PEave /= (steps/500);
                probacceptave /= (steps/500);
869
                // adjust the alpha value as try to make probacceptave to be in
871             // between 0.2 and 0.4, also alpha can't be greater than 5
                // otherwise the block will be blow up
873             if(probacceptave > 0.35 && alpha < 5){
                    // increase alpha
875                 alpha*=1.05;
                }
877             else if (probacceptave <0.25){
                    //decrease alpha
879                 alpha/=1.05;
                }
881
                cout <<"Ave alpha = \t"<<sumalpha<<"Accept prob \t: "<<
    probacceptave<<"PE \t"
883                 <<PEave<<"PEnew \t "<<newPE<<"prob \t"<<prob<<endl;
                counter = 0;
885             st <<PEave<<" "<<probacceptave<<" "<<sumalpha<<endl;
            }
887
            // record the energy data:
889         // st << PE <<" "<<probaccept<<endl;
            counter++;
891         it++;
        }
893     // last update the force and stress:
        get_atomic_stress();
895
        string filename = dir +"/Energy.txt";
897     ofstream myfile(filename);
        string data = st.str();
899     myfile<< data;
        myfile.close();
901 }

903 void block::CollectEqdata(long int steps,double T,string dir){
        stringstream st;
905     long int it = 0;
        double KE = 400*k_b*T;
907     double PE = 0;
        double alpha = 1;
909     double PEave = 0;
        double Pave = 0;
911     int counter = 0;
        double sumalpha = 0;
913     double probacceptave = 0;
        while(it<1E4){
915         PE = total_penergy();
            double oriXpos,oriYpos,newPE,probaccept;
917         double prob;
            int sidx;
919         // move the atom
```

```
        Move(alpha,sidx,oriXpos,oriYpos);

        // once we change the config, we update neighborlist and the PE
        get_neighbor_list(neighCut);
        newPE = total_penergy();

        // check the acceptance prob;
        if(newPE>PE){
            probaccept = exp(-(newPE-PE)/(k_b*T));
            prob = (double)rand()/ RAND_MAX;

            if(prob > probaccept){
                // do not accept if prob > probaccept
                redoMove(sidx,oriXpos,oriYpos);
            }
            else{
            // else accept move
                PE = newPE;
            }
        }
        else{
            probaccept = 1;
            // accept the move
            PE = newPE;
        }

        sumalpha += alpha;
        probacceptave += probaccept;

        if(counter > steps/100){
            sumalpha /=(steps/100);
            probacceptave /= (steps/100);

            // adjust the alpha value as try to make probacceptave to be in
            // between 0.2 and 0.4, also alpha can't be greater than 5
            // otherwise the block will be blow up
            if(probacceptave > 0.35 && alpha < 5){
                // increase alpha
                alpha*=1.05;
            }
            else if (probacceptave <0.25){
                //decrease alpha
                alpha/=1.05;
            }
        }

        PEave += PE;
        get_atomic_stress();
        array<double,4> stress = calc_total_stress_pressure();
        Pave += stress[3];
        counter++;
        it++;
    }
    // last update the force and stress:
```

```
    PEave /= steps;
    Pave /= steps;
    st<< PEave <<" "<<KE<<" "<<Pave<<endl;
    cout<< "done" + dir << endl;

    string filename = dir +"/Equilibrium.txt";
    ofstream myfile(filename);
    string data = st.str();
    myfile<< data;
    myfile.close();

}


///------------------------------Outside block class ------------------
double psi(double r){
    double result;
    //  r >= rCut
    if(r>=rCut){result = 0;}
    //  rTail <= r < rCut
    else if(r>=rTail){result = A*pow(r-rCut,3) + B*pow(r-rCut,2);}
    //  r < rTail
    else{result = 4.0*epsilon*(pow(sigma/r,12) - pow(sigma/r,6));}

    return result;
}

double dpsi(double r){
    double result;
    if(r>=rCut){result = 0;}
    else if (r>= rTail){result = 3.0*A*pow(r-rCut,2) + 2.0*B*(r-rCut); }
    else{result = (24.0*epsilon/r)*(pow(sigma/r,6)-2.0*pow(sigma/r,12));}
    return result;
}
```

Listing 2: 2dblock.cpp

```
// main.cpp
// HW5 Monte Carlo Simulation
// Main function
// Author: Yuding Ai
// Penn ID: 31295008
// Data: 2017.04.11

#include "2dblock.h"

int main(){
    double start = clock();
    block bloc1(20,20,400);

    //---------T = 5K ----------------------
    bloc1.MonteCarlo(5E5, 5, "5K");
    bloc1.out_config("5K/minimization.txt");
    bloc1.rdf("5K/rdf_MDmini.txt");
```

```
    //----------T = 10K --------------------
    bloc1.set_config("5K/minimization.txt");
    bloc1.MonteCarlo(5E5, 10, "10K");
    bloc1.out_config("10K/minimization.txt");
    bloc1.rdf("10K/rdf_MDmini.txt");

    //----------T = 15K --------------------
    bloc1.set_config("10K/minimization.txt");
    bloc1.MonteCarlo(5E5, 15, "15K");
    bloc1.out_config("15K/minimization.txt");
    bloc1.rdf("15K/rdf_MDmini.txt");

    //----------T = 20K --------------------
    bloc1.set_config("15K/minimization.txt");
    bloc1.MonteCarlo(5E5, 20, "20K");
    bloc1.out_config("20K/minimization.txt");
    bloc1.rdf("20K/rdf_MDmini.txt");

    //----------T = 25K --------------------
    bloc1.set_config("20K/minimization.txt");
    bloc1.MonteCarlo(5E5, 25, "25K");
    bloc1.out_config("25K/minimization.txt");
    bloc1.rdf("25K/rdf_MDmini.txt");

    //----------T = 30K --------------------
    bloc1.set_config("25K/minimization.txt");
    bloc1.MonteCarlo(5E5, 30, "30K");
    bloc1.out_config("30K/minimization.txt");
    bloc1.rdf("30K/rdf_MDmini.txt");

    //----------T = 35K --------------------
    bloc1.set_config("30K/minimization.txt");
    bloc1.MonteCarlo(5E5, 35, "35K");
    bloc1.out_config("35K/minimization.txt");
    bloc1.rdf("35K/rdf_MDmini.txt");

    //----------T = 40K --------------------
    bloc1.set_config("35K/minimization.txt");
    bloc1.MonteCarlo(5E5, 40, "40K");
    bloc1.out_config("40K/minimization.txt");
    bloc1.rdf("40K/rdf_MDmini.txt");

    //----------T = 45K --------------------
    bloc1.set_config("40K/minimization.txt");
    bloc1.MonteCarlo(5E5, 45, "45K");
    bloc1.out_config("45K/minimization.txt");
    bloc1.rdf("45K/rdf_MDmini.txt");

    //----------T = 50K --------------------
    bloc1.set_config("45K/minimization.txt");
    bloc1.MonteCarlo(5E5, 50, "50K");
    bloc1.out_config("50K/minimization.txt");
    bloc1.rdf("50K/rdf_MDmini.txt");
```

```
73      //----------T = 55K --------------------
        bloc1.set_config("50K/minimization.txt");
75      bloc1.MonteCarlo(5E5, 55, "55K");
        bloc1.out_config("55K/minimization.txt");
77      bloc1.rdf("55K/rdf_MDmini.txt");

79      //----------T = 60K --------------------
        bloc1.set_config("55K/minimization.txt");
81      bloc1.MonteCarlo(5E5, 60, "60K");
        bloc1.out_config("60K/minimization.txt");
83      bloc1.rdf("60K/rdf_MDmini.txt");

85      //----------T = 65K --------------------
        bloc1.set_config("60K/minimization.txt");
87      bloc1.MonteCarlo(5E5, 65, "65K");
        bloc1.out_config("65K/minimization.txt");
89      bloc1.rdf("65K/rdf_MDmini.txt");

91      //----------T = 70K --------------------
        bloc1.set_config("65K/minimization.txt");
93      bloc1.MonteCarlo(5E5, 70, "70K");
        bloc1.out_config("70K/minimization.txt");
95      bloc1.rdf("70K/rdf_MDmini.txt");

97      //----------T = 75K --------------------
        bloc1.set_config("70K/minimization.txt");
99      bloc1.MonteCarlo(5E5, 75, "75K");
        bloc1.out_config("75K/minimization.txt");
101     bloc1.rdf("75K/rdf_MDmini.txt");

103     //----------T = 80K --------------------
        bloc1.set_config("75K/minimization.txt");
105     bloc1.MonteCarlo(5E5, 80, "80K");
        bloc1.out_config("80K/minimization.txt");
107     bloc1.rdf("80K/rdf_MDmini.txt");

109

        // --- Now collect the equalibrium info from each state----
111     // --- Corresponding to each temperature ------------------

113     // ------------T = 5K --------------------
        bloc1.set_config("5K/minimization.txt");
115     bloc1.CollectEqdata(1E4, 5, "5K");

117     // ------------T = 10K --------------------
        bloc1.set_config("10K/minimization.txt");
119     bloc1.CollectEqdata(1E4, 10, "10K");
        // ------------T = 15K --------------------
121     bloc1.set_config("15K/minimization.txt");
        bloc1.CollectEqdata(1E4, 15, "15K");
123     // ------------T = 20K --------------------
        bloc1.set_config("20K/minimization.txt");
125     bloc1.CollectEqdata(1E4, 20, "20K");
```

```
    // ------------T = 25K --------------------
127     bloc1.set_config("25K/minimization.txt");
    bloc1.CollectEqdata(1E4,25, "25K");
129     // ------------T = 30K --------------------
    bloc1.set_config("30K/minimization.txt");
131     bloc1.CollectEqdata(1E4,30, "30K");
    // ------------T = 35K --------------------
133     bloc1.set_config("35K/minimization.txt");
    bloc1.CollectEqdata(1E4, 35, "35K");
135     // ------------T = 40K --------------------
    bloc1.set_config("40K/minimization.txt");
137     bloc1.CollectEqdata(1E4, 40, "40K");
    // ------------T = 45K --------------------
139     bloc1.set_config("45K/minimization.txt");
    bloc1.CollectEqdata(1E4, 45, "45K");
141     // ------------T = 50K --------------------
    bloc1.set_config("50K/minimization.txt");
143     bloc1.CollectEqdata(1E4, 50, "50K");
    // ------------T = 55K --------------------
145     bloc1.set_config("55K/minimization.txt");
    bloc1.CollectEqdata(1E4, 55, "55K");
147     // ------------T = 60K --------------------
    bloc1.set_config("60K/minimization.txt");
149     bloc1.CollectEqdata(1E4, 60, "60K");
    // ------------T = 65K --------------------
151     bloc1.set_config("65K/minimization.txt");
    bloc1.CollectEqdata(1E4, 65, "65K");
153     // ------------T = 70K --------------------
    bloc1.set_config("70K/minimization.txt");
155     bloc1.CollectEqdata(1E4, 70, "70K");
    // ------------T = 75K --------------------
157     bloc1.set_config("75K/minimization.txt");
    bloc1.CollectEqdata(1E4,75, "75K");
159     // ------------T = 80K --------------------
    bloc1.set_config("80K/minimization.txt");
161     bloc1.CollectEqdata(1E4, 80, "80K");

163     double end = clock();
    cout <<"This simulation takes: "<< (double(end-start)/CLOCKS_PER_SEC)<<"
    sec."<<endl;
165
    return 0;
167 }
```

Listing 3: main.cpp

```
1 # hw5.py
  # HW5 Monte Carlo Simulation
3 # Author: Yuding Ai
  # Date: 2017.04.11
5
  import math
7 import numpy as np
  import matplotlib.mlab as mlab
```

```
import matplotlib.pyplot as plt
import scipy.stats  as stats
import collections
import matplotlib as mpl
from matplotlib import rc
from itertools import groupby
rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)

def plotconfig(txtname,filename):
    '''take the config.txt to plot the atoms'''
    X = [] # a list of Xpos
    Y = [] # a list of Ypos
    Sxx = [] # a list of signa_xx
    Syy = [] # a list of signa_yy
    Sxy = [] # a list of signa_xy
    P = []
    with open(txtname,"r") as file:
        for line in file:
            words = line.split()
            x = float(words[0]) #take the value
            y = float(words[1]) #take the value
            sxx = float(words[2]) #take the value
            syy = float(words[3]) #take the value
            sxy = float(words[4]) #take the value
            p = -0.5*(syy+sxx)

            X.append(x); #append x value into X
            Y.append(y); #append y value into Y
            Sxx.append(sxx);
            Syy.append(syy);
            Sxy.append(sxy);
            P.append(p);

    fig = plt.figure()
    plt.plot(X,Y,'ro',linewidth = 0.8)
    plt.ylim([-4,80])
    plt.xlim([-4,80])
    xlabel = r'$x_{coordinate}\ unit =\AA $'
    ylabel = r'$y_{coordinate}\ unit =\AA $'
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title('Configuration of Atoms')
    fig.savefig(filename,dpi = 300, bbox_inches ='tight')

    #Plot the stresses as heatmap
    #---------------simgaxx--------------------------------
    fig, ax = plt.subplots(1)
    plt.scatter(X,Y,s = 150,marker = '8', c = Sxx,alpha=0.7)
    cbar = plt.colorbar()
    cbar.set_label(r"$eV/{\AA}^2$")
    plt.clim(-0.006,0.008)
    xlabel = r'X_{pos}'
    ylabel = r'Y_{pos}'
```

```
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.locator_params(axis='y', nticks=3)
    plt.locator_params(axis='x', nticks=8)
    fname = "sigma_xx" + filename
    titlename = r"$\sigma_{xx}$"
    plt.title(titlename)
    fig.savefig(fname,dpi=300)

    #---------------simgayy---------------------------------
    fig, ax = plt.subplots(1)
    plt.scatter(X,Y,s = 150,marker = '8', c = Syy,alpha=0.7)
    cbar = plt.colorbar()
    cbar.set_label(r"$eV/{\AA}^2$")
    plt.clim(-0.006,0.008)
    xlabel = r'X_{pos}'
    ylabel = r'Y_{pos}'
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.locator_params(axis='y', nticks=3)
    plt.locator_params(axis='x', nticks=8)
    fname = "sigma_yy" + filename
    titlename = r"$\sigma_{yy}$"
    plt.title(titlename)
    fig.savefig(fname,dpi=300)

    #---------------simgaxy---------------------------------
    fig, ax = plt.subplots(1)
    plt.scatter(X,Y,s = 150,marker = '8', c = Sxy,alpha=0.7)
    cbar = plt.colorbar()
    cbar.set_label(r"$eV/{\AA}^2$")
    plt.clim(-0.006,0.008)
    xlabel = r'X_{pos}'
    ylabel = r'Y_{pos}'
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.locator_params(axis='y', nticks=3)
    plt.locator_params(axis='x', nticks=8)
    fname = "sigma_xy" + filename
    titlename = r"$\sigma_{xy}$"
    plt.title(titlename)
    fig.savefig(fname,dpi=300)

    #---------------  P ----------------------------------
    fig, ax = plt.subplots(1)
    plt.scatter(X,Y,s = 150,marker = '8', c = P,alpha=0.7)
    cbar = plt.colorbar()
    cbar.set_label(r"$eV/{\AA}^2$")
    pmax = max(P)
    pmin = min(P)
    if pmax > -pmin:
        plt.clim(-pmax,pmax)
    else:
        plt.clim(pmin,-pmax)
```

```
117
    xlabel = r'X_{pos}'
119     ylabel = r'Y_{pos}'
    plt.xlabel(xlabel)
121     plt.ylabel(ylabel)
    plt.locator_params(axis='y', nticks=3)
123     plt.locator_params(axis='x', nticks=8)
    fname = "hydrop" + filename
125     titlename = r"Hydrostatic pressure"
    plt.title(titlename)
127     fig.savefig(fname,dpi=300)

129
def E_vs_it():
131     PE = [] # a list of PE
    IT = [] # Iteration
133     AC = [] # Accept PRob
    with open("Energy.txt","r") as file:
135         it = 0
        for line in file:
137             words = line.split()
            pe = float(words[0]) #take the value
139             ac = float(words[1]) #take the value
            it = it + 1
141
            AC.append(ac);
143             PE.append(pe);
            IT.append(it);
145
    fig = plt.figure()
147     plt.plot(IT,AC,'ro-',linewidth = 0.5,markersize = 1,label = r'Accept Prob'
    )
    leg = plt.legend(prop={'size':10})
149     leg.get_frame().set_alpha(0.5)
    xlabel = r'Iterations (1000 steps per Iteration)'
151     ylabel = r'Movement acceptance Probability'
    plt.ylim([0,1])
153     plt.xlabel(xlabel)
    plt.ylabel(ylabel)
155     plt.title(r'Acceptance Prob vs Iteration')
    fig.savefig("ACvsIT",dpi = 300, bbox_inches ='tight')
157


159     fig = plt.figure()
    plt.plot(IT,PE,'bo-',linewidth = 0.5,markersize = 1,label = r'PE')
161     leg = plt.legend(prop={'size':10})
    leg.get_frame().set_alpha(0.5)
163     xlabel = r'Iterations (1000 steps per Iteration)'
    ylabel = r'Potential Energy (eV)'
165     plt.ylim([-12,-5])
    plt.xlabel(xlabel)
167     plt.ylabel(ylabel)
    plt.title(r'Potential Energy vs Iteration')
169     fig.savefig("PEvsIT",dpi = 300, bbox_inches ='tight')
```

```
171 def plottemp_pressure(filename):
    T = [] # a list of KE
173     P = [] # a list of PE
    step = []
175     s = 0
    with open(filename,"r") as file:
177         for line in file:
            words = line.split()
179             t = float(words[0]) #take the value
            p = float(words[5]) #take the value
181             s= s+ 1

183             T.append(t);
            P.append(p);
185             step.append(s)

187     avep = sum(P)/len(P)
    print avep
189
    fig = plt.figure()
191     plt.plot(step,T,'go-',linewidth = 0.5,markersize = 5,label = r'Temperature
    ')
    leg = plt.legend(prop={'size':10})
193     leg.get_frame().set_alpha(0.5)
    xlabel = r'$N^{th}$ MD simulation'
195     ylabel = r'Temperature (K)'
    plt.xlabel(xlabel)
197     plt.ylabel(ylabel)
    plt.title(r'Temperature vs MD')
199     fig.savefig("temperature",dpi = 300, bbox_inches ='tight')

201     fig = plt.figure()
    plt.plot(step,P,'co-',linewidth = 0.5,markersize = 5,label = r'Pressure')
203     leg = plt.legend(prop={'size':10})
    leg.get_frame().set_alpha(0.5)
205     xlabel = r'$N^{th}$ MD simulation'
    ylabel = r'Pressure ($eV/\AA^2$)'
207     plt.xlabel(xlabel)
    plt.ylabel(ylabel)
209     plt.title(r'Pressure vs MD')
    fig.savefig("Pressure",dpi = 300, bbox_inches ='tight')
211


213
def main():
215     E_vs_it()
    plotconfig("minimization.txt","config.png")
217
main()
```

Listing 4: hw5.py

```
# rdf.py
```

```
# HW5 Monte Carlo Simulation
# Author: Yuding Ai
# Date: 2017.04.11

import math
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt
import scipy.stats  as stats
import collections
import matplotlib as mpl
from matplotlib import rc
from scipy.interpolate import spline
from itertools import groupby
rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)

def plotrdf(txtname,filename):
    '''take the rdf.txt to plot the rdf function'''

    R = [] # a list of distance distribution
    G = [] # a list of distance distribution
    with open(txtname,"r") as file:
        for line in file:
            words = line.split()
            r = float(words[0]) #take the value
            g = float(words[1]) #take the value
            R.append(r); #append r value into r
            G.append(g); #append g(r) value into G
    fig = plt.figure()

    x_sm = np.array(R)
    y_sm = np.array(G)
    x_smooth = np.linspace(x_sm.min(), x_sm.max(), 300)
    y_smooth = spline(R, G, x_smooth)
    # Create a canvas to place the subgraphs
    canvas = plt.figure()
    rect = canvas.patch
    rect.set_facecolor('white')
    sp1 = canvas.add_subplot(1,1,1, axisbg='w')

    if txtname == "rdf.txt":
        plt.stem(R, G, linefmt='b-', linewidth = 0.4,markerfmt='rx', basefmt='
   r-',label = r'RDF')
    else:
        sp1.plot(x_smooth, y_smooth, 'red', linewidth=1)
    # Colorcode the tick tabs
    sp1.tick_params(axis='x', colors='black')
    sp1.tick_params(axis='y', colors='black')

    xlabel = r'$r\ unit =\AA $'
    ylabel = r'$g(r)\ $'
    plt.title('Radial distribution function g(r)')
    plt.xlabel(xlabel)
```

```
    plt.ylabel(ylabel)
    plt.ylim([0,0.3])
    # Show the plot/image
    plt.tight_layout()
    plt.grid(alpha=0.8)
    plt.savefig(filename,dpi = 300, bbox_inches ='tight')


def main():
    plotrdf("rdf_MDmini.txt","rdf.png")

main()
```

Listing 5: rdf.py