Homework 4
Molecular Dynamics

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

April 3, 2017

1 Minimizing the Potential energy using MD

First of all, we minimize the potential energy and instead of molecular static approach, this time we use molecular dynamics. In this work, we will employ the so called **Verlet algotithm** to solve the equation of motion such that:

$$r_i((J+1)\Delta t) = r_i(J\Delta t) + v_i(J\Delta t)\Delta t + \frac{\Delta t^2}{2m_i}F_i(J\Delta t) \tag{1}$$

$$v_i((J+1)\Delta t) = v_i(J\Delta t) + \frac{\Delta t}{2m_i}(F_i((J+1)\Delta t) + F_i(J\Delta t)) \tag{2}$$

Utilizing this algorithm, we minimizing the potential energy by preforming a series Molecular dynamics simulations successively. After each MD simulation, we decrease the velocities by setting it back to zero and thus remove kinetic energy from the system. For each MD simulation, the total energy is conserved such that it is in a **micro-canonical ensemble**. As each of the MD simulation proceeds, the kinetic energy gradually increase into some threshold and the potential energy drops correspondingly. After a certain amount of MD simulations, finally the kinetic energy/velocities goes to zero and thus we obtained the minimized potential energy.

For our simulation, we choose Δt = 1e-14 and according to my result, 50000 iteration turns out to be an appropriate amount for each MD simulation. After **10 sets** of successive MD simulation, (before each MD, we reset velocities to 0 as to decrease the Temperature) we obtained the minimized potential energy.

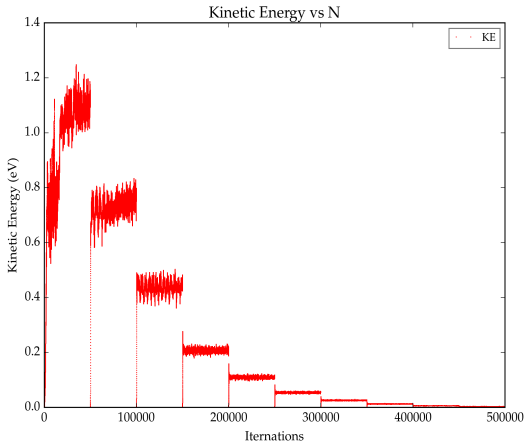
In short, the recipe for implementing this minimizing program could be written as a pseudocode in Algorithm1 (see detailed implementation in Appendix 2dblock.cpp, $MD\_mini()$)

Algorithm 1 Minimizing PE through Molecular dynamics

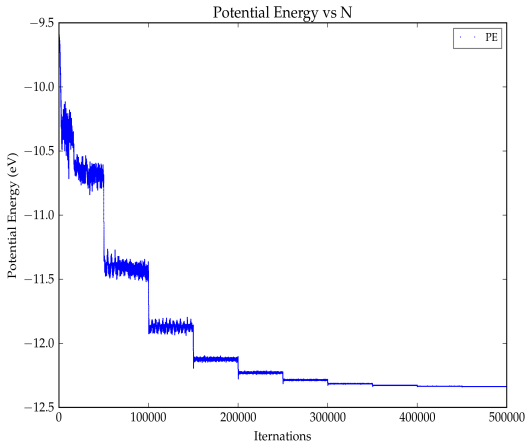
procedure 10 sets OF SUCCESSIVE MD SIMULATIONS
    reset velocities to 0
    Start from previous configurations
    while Each MD simulation with **50000 iterations do**
        record the current **Force on each atom** (it will be used to calculate new velocities)
        Move the atoms following Equation 1
        Update the neighbor list (since we moved the atoms in previous step)
        Calculate the new **Force on each atom** of the current configuration
        Calculate the new velocities using Equation 2 and update the atom's velocities
        Calculate and record the PE and KE of the current configuration
    Calculate and record the $< KE >$ and then obtain T
    Calculate and record the total hydrostatic pressure for the final configuration
Plot the final configuration and Radial distribution function

1.1 Minimizing PE result

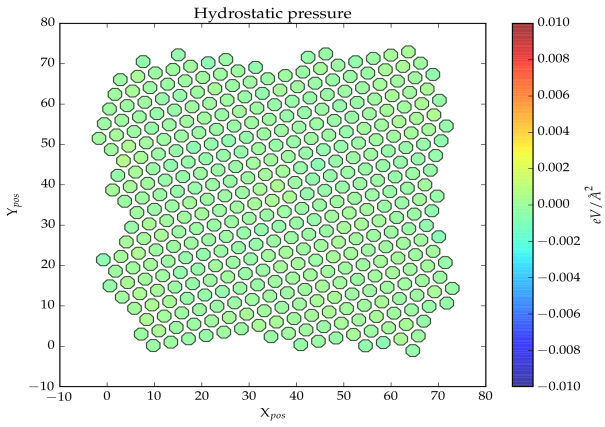
Here we display the minimization result in Figure 1. As is shown in Figure 1 (a) and (b), we see that the minimum potential is obtained after 10 sets of successive MD simulations which is **-12.3367 eV**. This is lower then the result from HW3 with Molecular static approach. At HW3, we got the lowest local minimum potential energy to be -11.7813 eV and corresponding to a polycrystalline structures with defects/grain boundaries. This time, as is shown in Figure 1 (c), the final structure is almost an ideal close packed crystal with no grain boundaries and all atoms has 0 hydrostatic pressure. As well as shown in the the RDF plot in (d), the 7 very sharp and thin peaks also indicates we have achieved a pretty good relaxed close packed crystal structure. Plot(e) and plot(f) are Total pressure and Temperature for the equilibrium state of each MD simulation and we see that the pressure is approaching to 0 at the final configuration (we got -0.0304906 eV/$\mathring{A}^2$)and the temperature is gradually decreased and approaching to 0 (**we got 0.0921K**, so it's not 0) as expected.



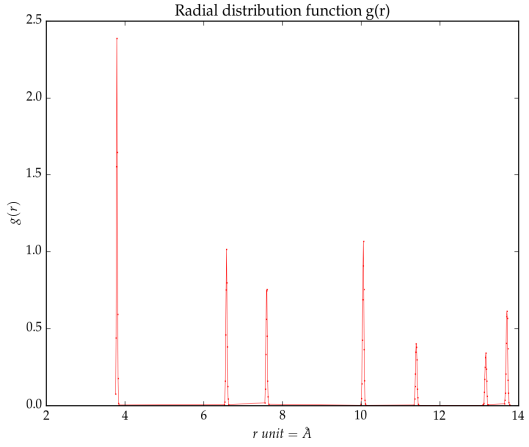
(a) Kinetic Energy vs Iterations



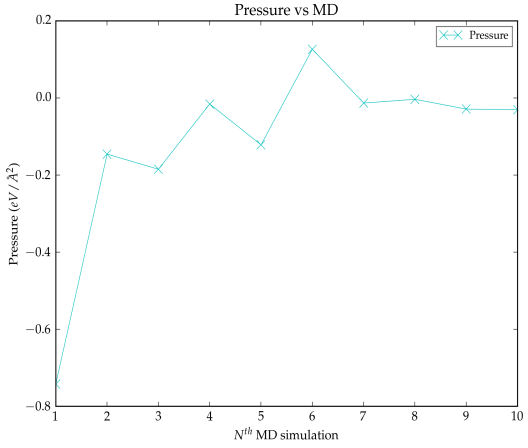
(b) Potential Energy vs Iterations



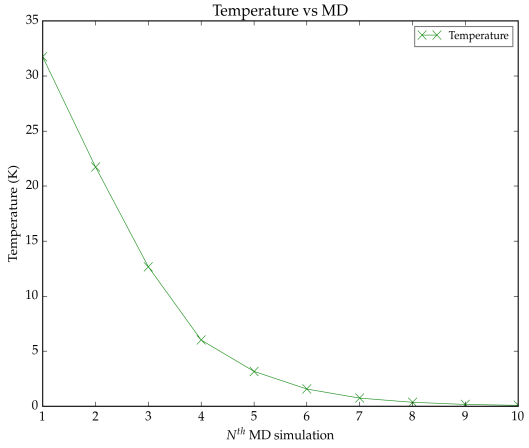
(c) Final configuration and hydrostatic pressure



(d) Final Radial distribution function



(e) Pressure vs MD simulations



(f) Temperature vs MD simulations

Figure 1: Minimizing PE by Molecular dynamics

2 Study the MD with different Temperatures

In this section, we study the Molecular dynamics in various temperatures. Starting with the relaxed close packed configuration obtained from the previous section, we gradually increase the Temperature from **0K to 80K in 16 steps**, with each step modifying the Temperature to increase by **5K**. We then study the configuration and the phase transition behavior of the block of atoms.

The temperature is adjusted by scaling the velocities as isokinetic Molecular dynamics such that:

$$v_i^{new} = \kappa \times v_i^{odd} \tag{3}$$

where κ is the scale factor, in 2D, κ is defined as

$$\kappa = [\frac{2Nk_BT_{desired}}{\sum_{i=1}^{N} m_i < v_i^2 >}]^{1/2} \tag{4}$$

According to equipartition thereon, for a 2D block of atoms

$$Nk_BT = \frac{1}{2}\sum_{i=1}^{N} m_i < v_i^2 > \tag{5}$$

Hence that we could rewrite Equation 4 as

$$\kappa = [\frac{T_{desired}}{T}]^{1/2} \tag{6}$$

According to my simulation, 30 successive MD simulations with 1000 iterations each is sufficient enough to raise the temperature by 5K. Prior to each of the 30 MD simulation, we raise the Temperature using Equation 6, hence at the end of 30 MD simulations, κ approaches to 1 and T gets close to T_{desire}.

The recipe for implementing Molecular simulation for this part is quite similar to the one at last section, except for that instead of setting velocity to 0 before each MD simulation, this time we modify the velocity using Equation 6 as to make it get closer to the T_{desire}. The pseudocode for this section is depicted in Algorithm 2. (see detailed implementation in Appendix block MD()) Following this recipe, we preform 16 sets of such MD simulations thus raise the Temperature from 0K to 80K.

Algorithm 2 Molecular dynamics with various Temperatures

procedure 30 sets OF SUCCESSIVE MD SIMULATIONS
    Scale the velocity using Equation 6
    Start from previous configurations
    while 1000 iterations (for each MD simulations) **do**
        record the current **Force on each atom** (it will be used to calculate new velocities)
        Move the atoms following Equation 1
        Update the neighbor list (since we moved the atoms in previous step)
        Calculate the new **Force on each atom** of the current configuration
        Calculate the new velocities using Equation 2 and update the atom's velocities
        Calculate and record the PE and KE of the current configuration
    Calculate and record the $< KE >$ and then obtain T
    Calculate and record the total hydrostatic pressure for the final configuration
Plot the final configuration and Radial distribution function
Plot PE vs Iterations and KE vs Iterations

2.1 Result of MD with different Temperature

2.1.1 Configuration, RDF, Pressure and Temperature

The output of the simulation for each temperature is displayed at next few pages from Figure 2 to Figure 17.

We first plot the final configuration and the corresponding radial distribution function for each Temperature as well as plotting Energy vs Iterations, Pressure and Temperature vs MD simulations. Then using this final configuration (which has reached equilibrium), we continue to run another simulation as to find the autocorrelation function for each temperature. Last, when the phase shift from solid to liquid, in other words, when the diffusion took place, which is from 50K in our case, we then run another simulation as to find the self-diffusion coefficient and plot lnD vs 1/T.

As is shown in the output plots, we see that from 5K to 30K, the configuration of the atoms has not changed much while the hydrostatic pressure is gradually decreased. Although the radial distribution function became more and more fuzzy 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, it starts to have atom 'escape' from the block and fly away with 0 pressure on it and this is a sign that the phase transformation 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 at 75K and 80K**, from the final configuration plot (plot (b)), we see there are many sparsely distributed atoms surrounding around the 'core' (liquid phase) of the block, and **those atom are possibly formed a gas phase of the material**.

2.1.2 E vs Iterations and Heat capacity

Using the data collected through our simulations, we can then plot Potential energy vs Temperatures and Kinetic energy vs Temperatures where both PE and KE are the average value calculated through each set of the MD simulations after the equilibrium has been reached. Furthermore, Sum over KE and PE, we plot the total energy vs Temperature. Since we have a free surface, the pressure from outside is 0, thus that we could calculate the heat capacity C_p by its definition:

$$C_p = (\frac{\partial H}{\partial T})_p \approx (\frac{\partial E_{tot}}{\partial T})_p \tag{7}$$

where we have assumed that the Volume change is very small. (if we only interested in the 'liquid' phase after phase transformation so that the volume is still approximate to about 8100 $\mathring{A}^2$) In such case, we could neglect the small volume change so that $(\frac{\partial H}{\partial T})_p \approx (\frac{\partial E_{tot}}{\partial T})_p$.

As is shown in Figure 18, both KE and PE gradually increased as Temperature increased. **The Total Energy increase gradually even when the phase transformation took place and not showing a jump as a signature of first order transformation, this is due to the size of our model is too small that merely 400 atoms are not sufficient enough to show such 'big jump'**. On the other hand, as is expected, **the Heat capacity plot shows a peak (although not very sharp, due to the same reason of the model size) at around 40K-60K, which is another signature that the phase transformation has taken place over this temperature range**.

2.1.3 Autocorrelation function of velocity and self-diffusion coefficient D

Lastly, once we succeed in observing the phase transformation behavior, we can study the autocorrelation function of velocity and the self-diffusion coefficient D for the liquid/gas state.
In simulation, the autocorrelation function could be computed as:

$$< A(t)A(0) >= \frac{1}{M}\sum_{J=1}^{M} A(t + J\Delta t)A(J\Delta t) \tag{8}$$

and thus for our 400 atoms block, we could calculate the autocorrelation function for velocity by:

$$< v(t)v(0) >= \frac{1}{MN}\sum_{J=1}^{M}\sum_{i=1}^{N} v_i(t + J\Delta t)v_i(J\Delta t) \tag{9}$$

where N is the number of atoms, so 400 in this case.

To study the self-diffusion coefficient D, we use the Einstein's relation such that:

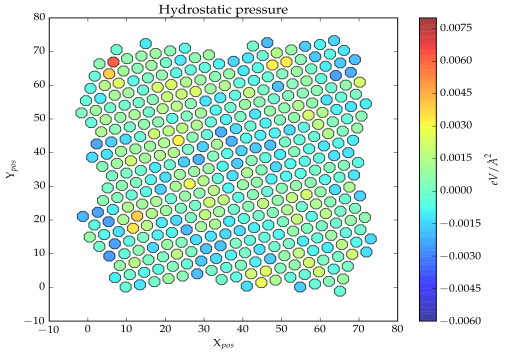
$$D = \lim_{\tau\to\infty}\frac{1}{6\tau} < [r(\tau) - r(0)]^2 > \tag{10}$$

Employ the same idea as for calculating the autocorrelation function, for our 400 atoms block, we could obtain the D value through simulation by:

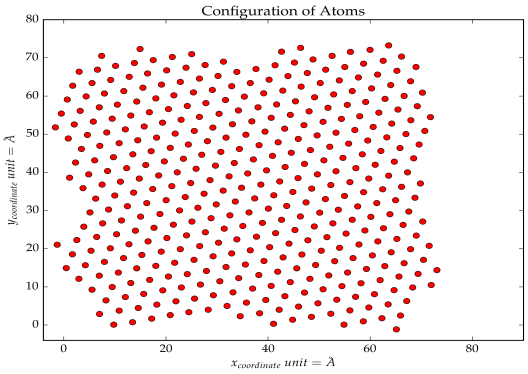
$$D = \lim_{\tau \to \infty} \frac{1}{6\tau} \sum_{J=1}^{M} \sum_{i=1}^{N} [r_i(\tau + J\Delta t) - r_i(J\Delta t)]^2 \tag{11}$$

where τ does not has to be ∞, rather, it is finite as long as $\tau >> t_{cor}$. In our case, according to the autocorrelation plots, t_{cor} is about 200 fs to 500 fs so we choose $\tau = 5000fs$. See detailed implementation at Appendix: 2dblock.cpp, autocorrelation() and diffusion().

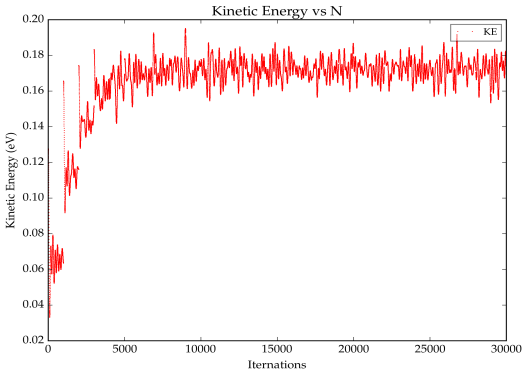
Finally we plot lnD vs 1/T at Figure 18 (f). From the lnD vs 1/T plot, we see that lnD is linearly depend on 1/T, which agrees with the theory since $lnD = lnD_o + \frac{-\Delta E}{k_B T}$. According to our data, the slope is about -213.33K, hence the corresponding activation energy $\Delta E = k_B * 213.33K = 0.018eV$



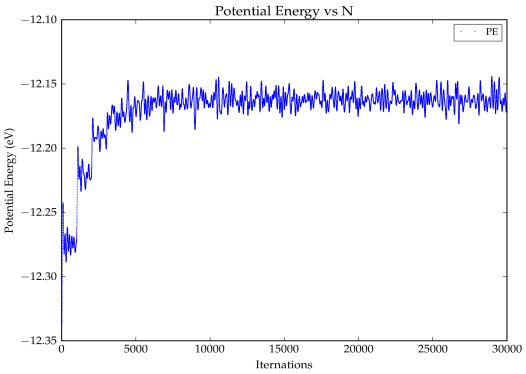
(a) Final Configuration and hydrostatic pressure



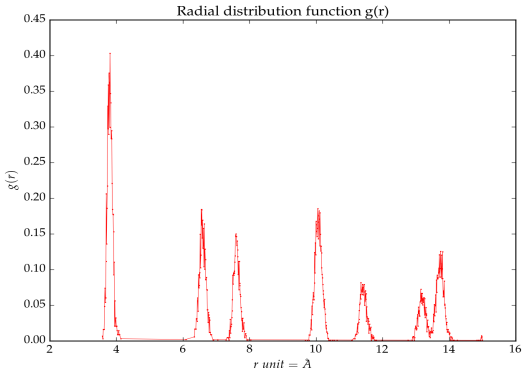
(b) Final Configuration



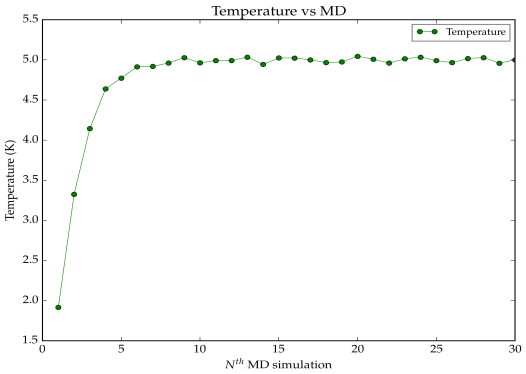
(c) Kinetic energy vs iteration



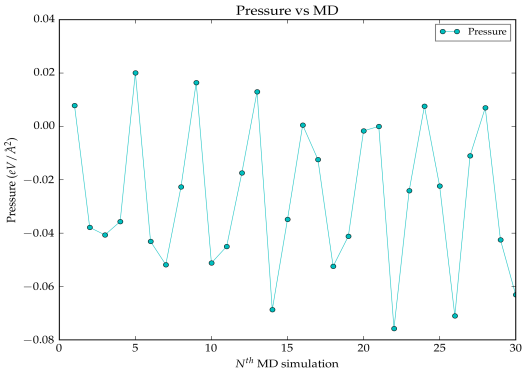
(d) Potential energy vs iteration



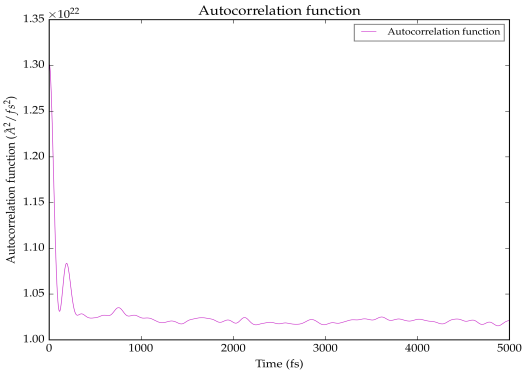
(e) Final Radial distribution function



(f) Temperature vs iteration

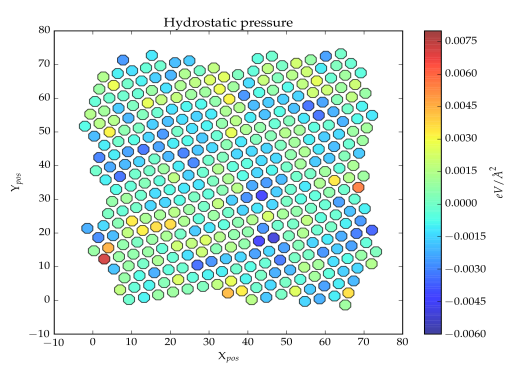


(g) Hydrostatic Pressure vs iteration

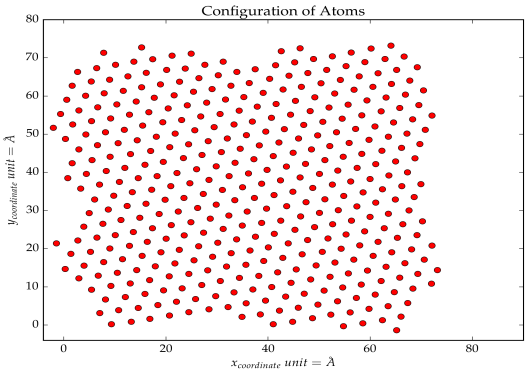


(h) Autocorrelation function

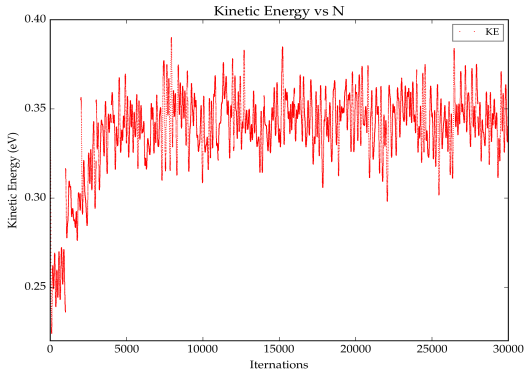
Figure 2: Temperature raise from 0K to 5K



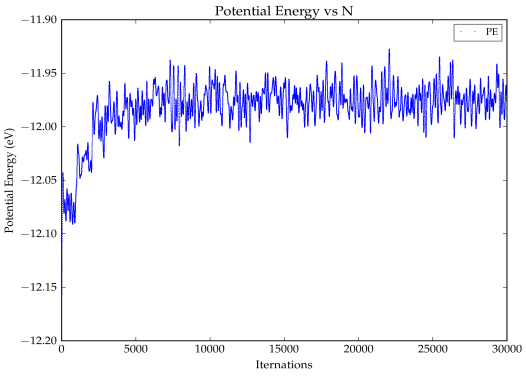
(a) Final Configuration and hydrostatic pressure



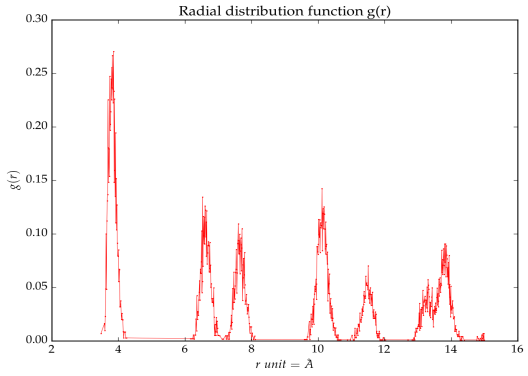
(b) Final Configuration



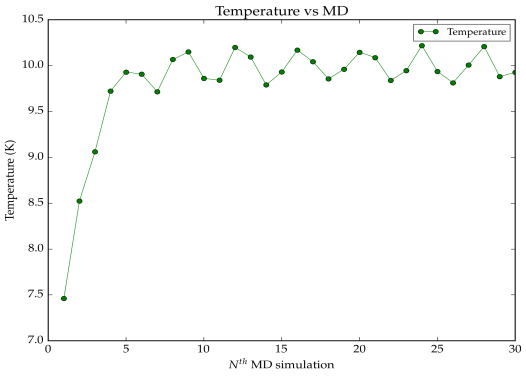
(c) Kinetic energy vs iteration



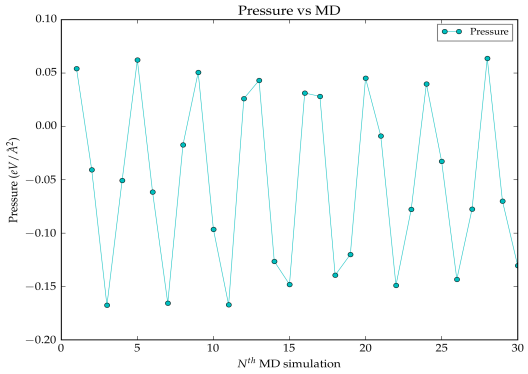
(d) Potential energy vs iteration



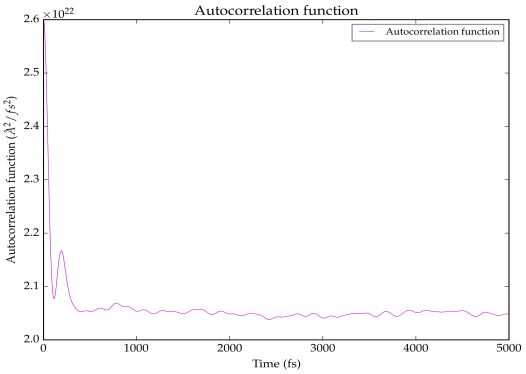
(e) Final Radial distribution function



(f) Temperature vs iteration

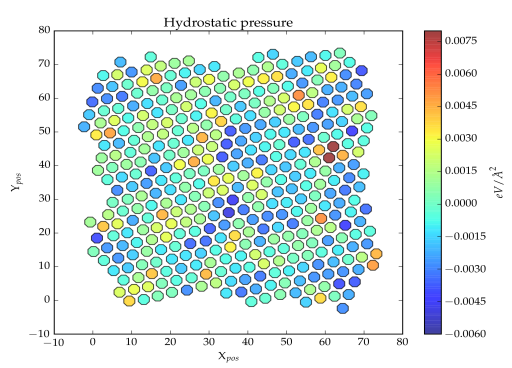


(g) Hydrostatic Pressure vs iteration

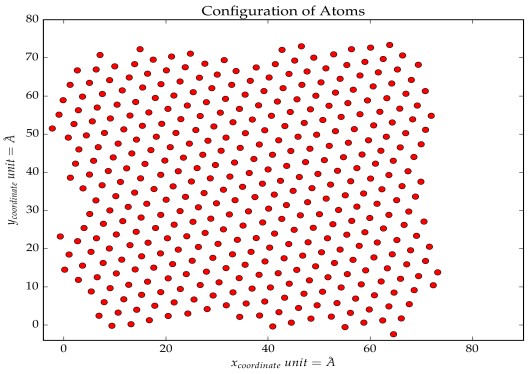


(h) Autocorrelation function

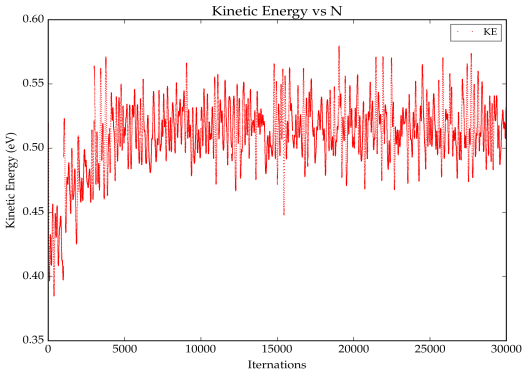
Figure 3: Temperature raise from 5K to 10K



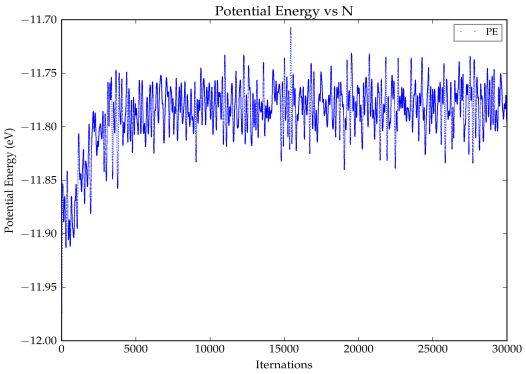
(a) Final Configuration and hydrostatic pressure



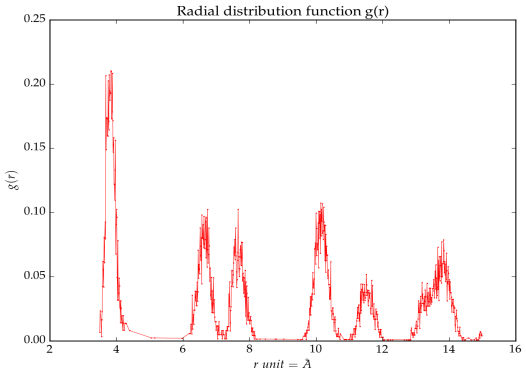
(b) Final Configuration



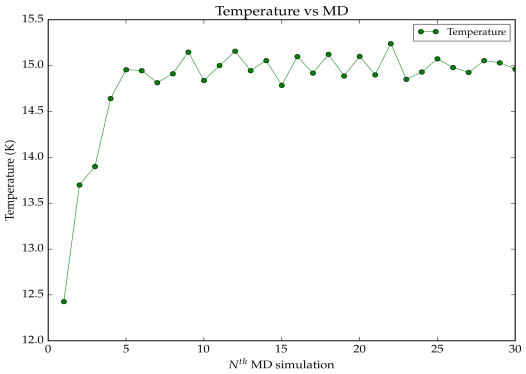
(c) Kinetic energy vs iteration



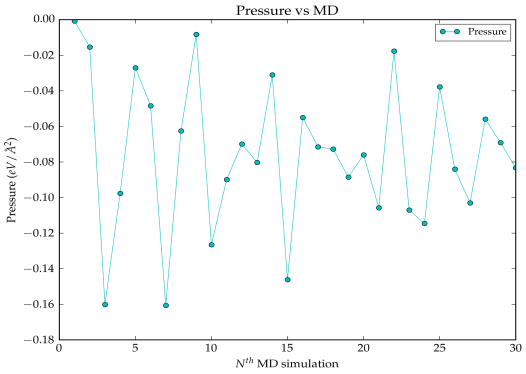
(d) Potential energy vs iteration



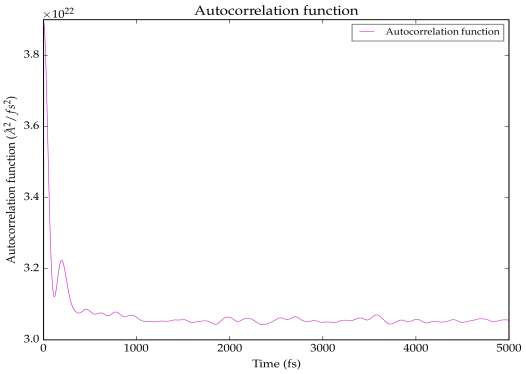
(e) Final Radial distribution function



(f) Temperature vs iteration

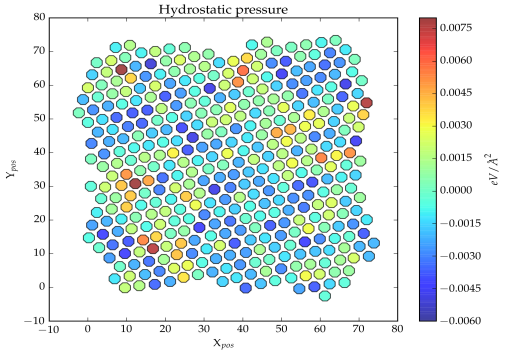


(g) Hydrostatic Pressure vs iteration

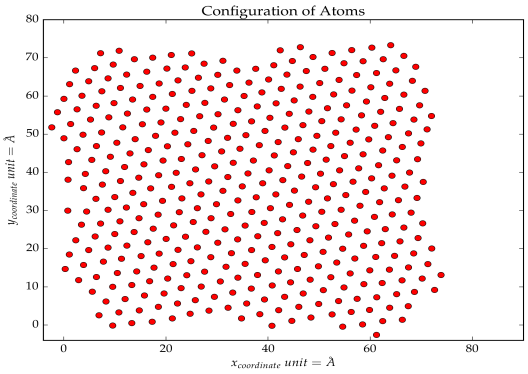


(h) Autocorrelation function

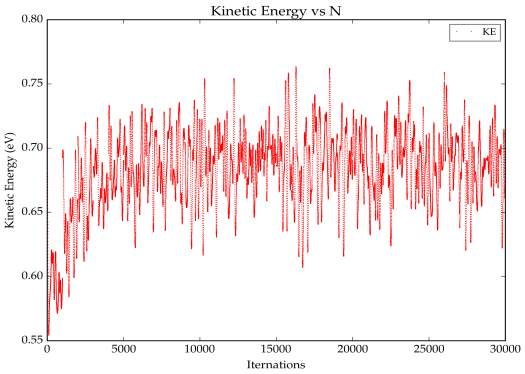
Figure 4: Temperature raise from 10K to 15K



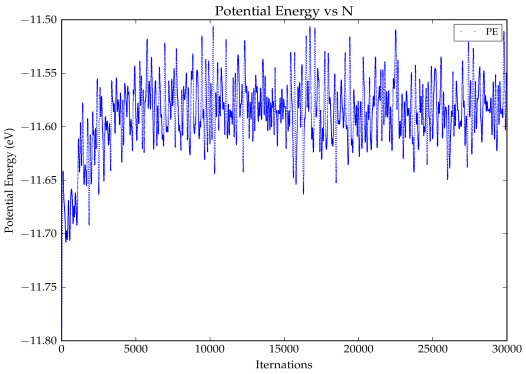
(a) Final Configuration and hydrostatic pressure



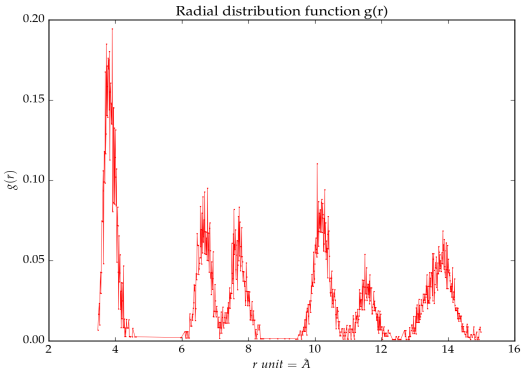
(b) Final Configuration



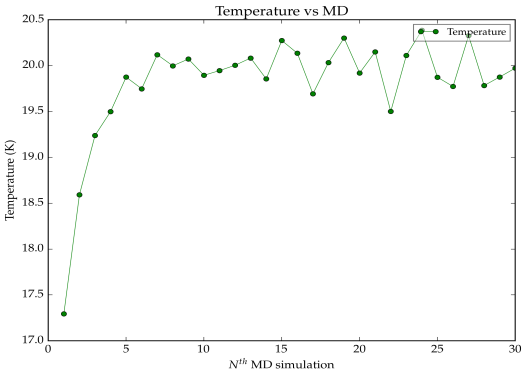
(c) Kinetic energy vs iteration



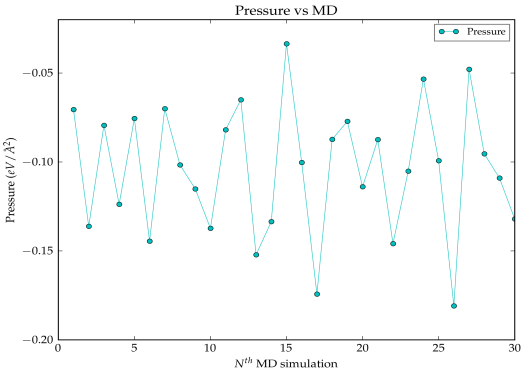
(d) Potential energy vs iteration



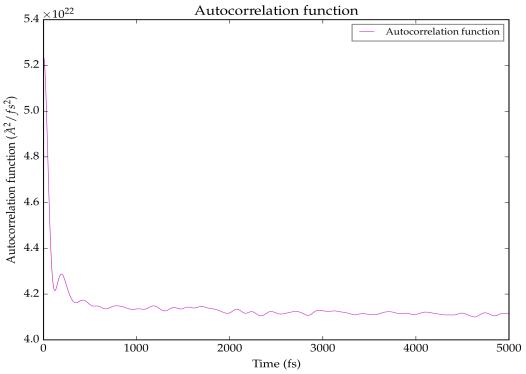
(e) Final Radial distribution function



(f) Temperature vs iteration

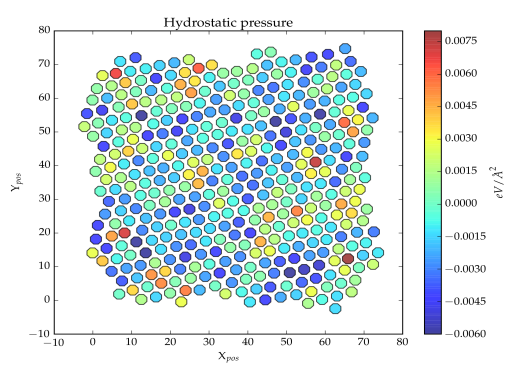


(g) Hydrostatic Pressure vs iteration

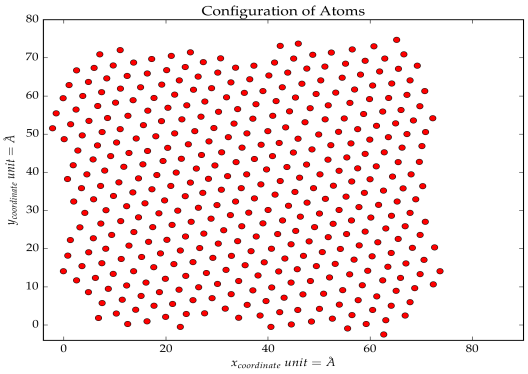


(h) Autocorrelation function

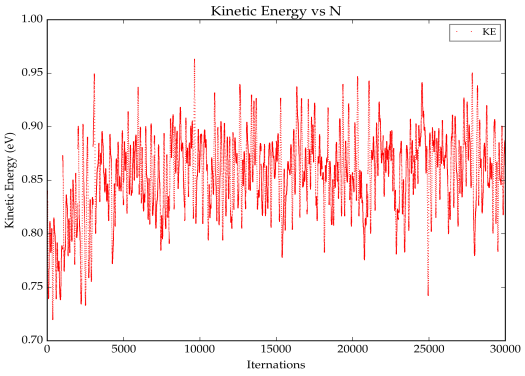
Figure 5: Temperature raise from 15K to 20K



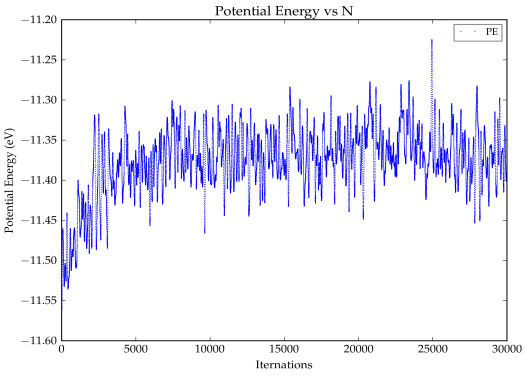
(a) Final Configuration and hydrostatic pressure



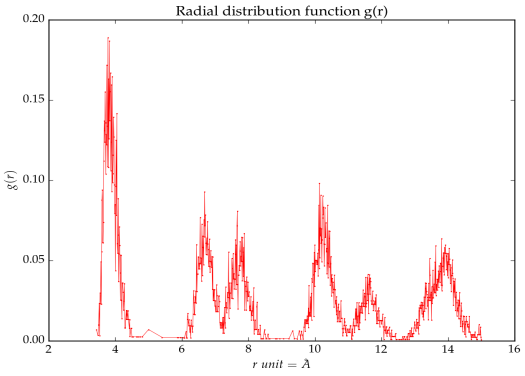
(b) Final Configuration



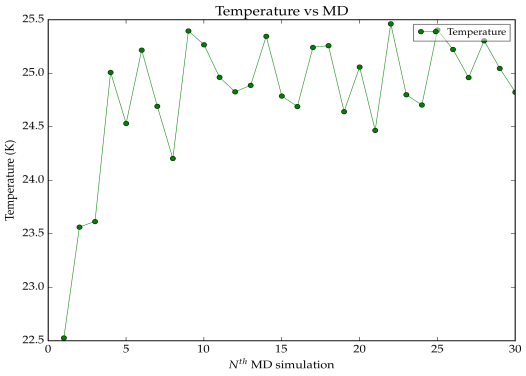
(c) Kinetic energy vs iteration



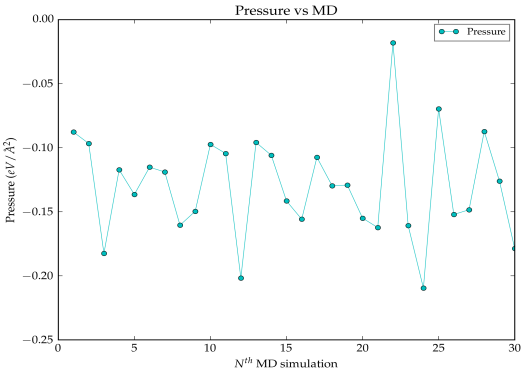
(d) Potential energy vs iteration



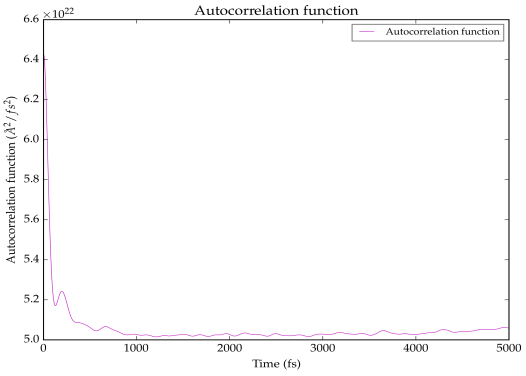
(e) Final Radial distribution function



(f) Temperature vs iteration

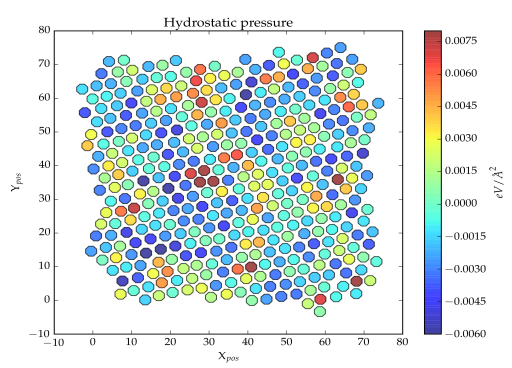


(g) Hydrostatic Pressure vs iteration

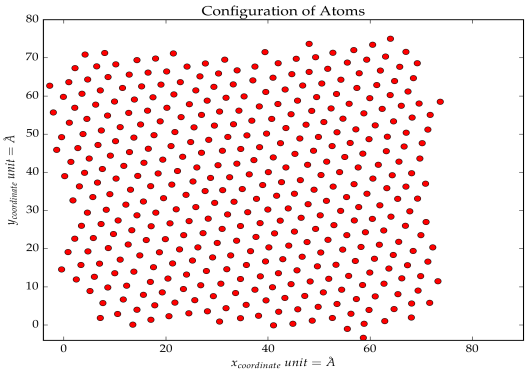


(h) Autocorrelation function

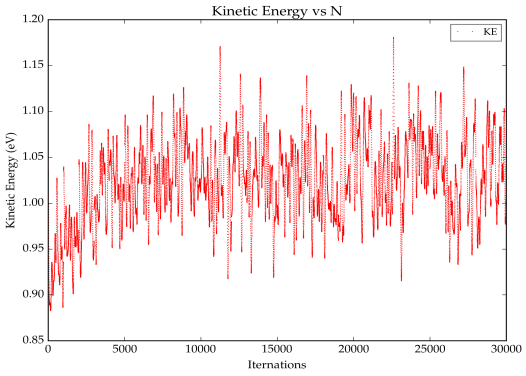
Figure 6: Temperature raise from 20K to 25K



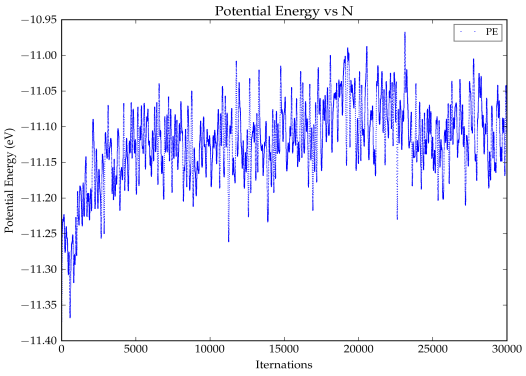
(a) Final Configuration and hydrostatic pressure



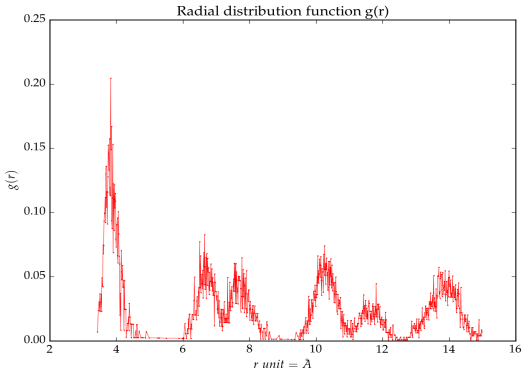
(b) Final Configuration



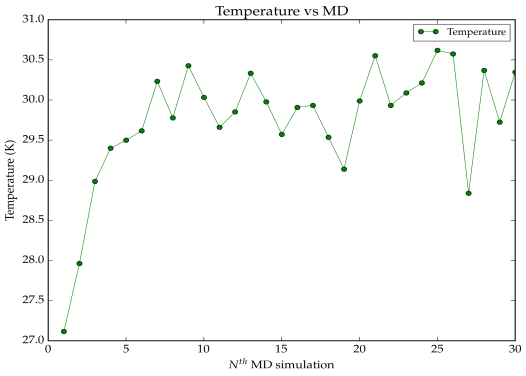
(c) Kinetic energy vs iteration



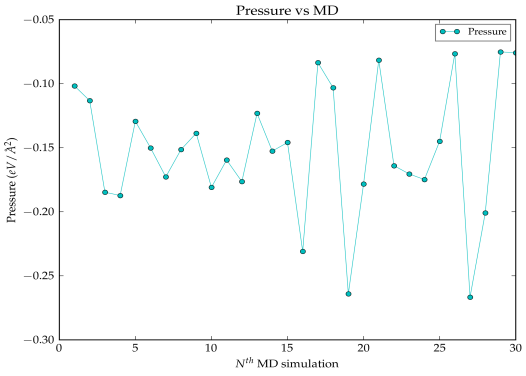
(d) Potential energy vs iteration



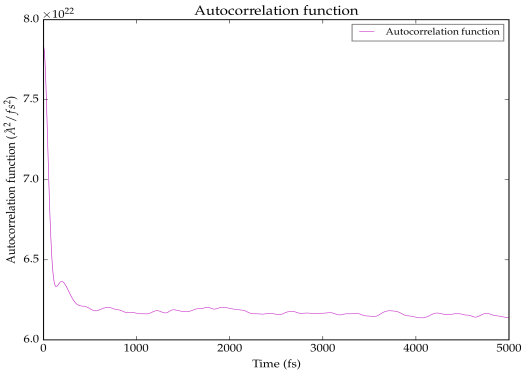
(e) Final Radial distribution function



(f) Temperature vs iteration

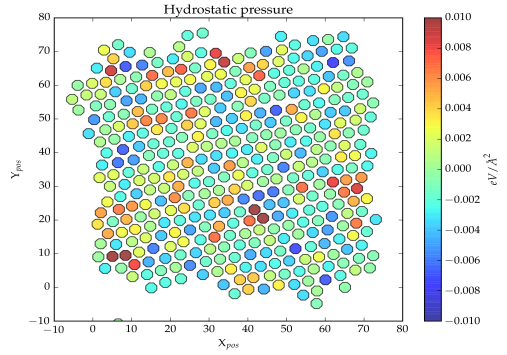


(g) Hydrostatic Pressure vs iteration

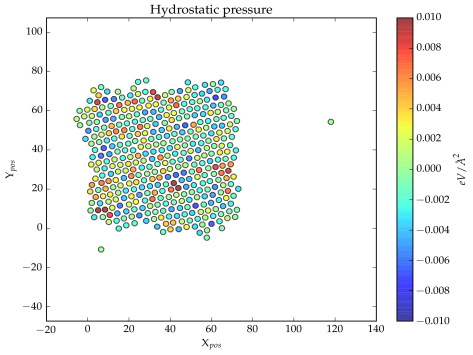


(h) Autocorrelation function

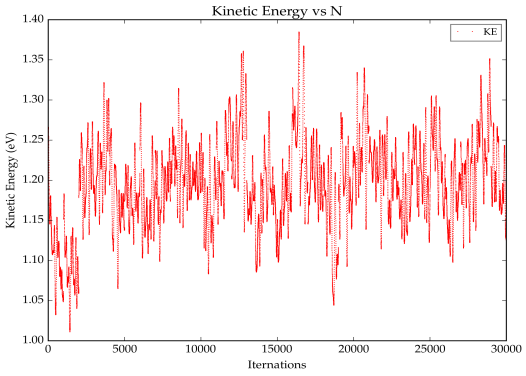
Figure 7: Temperature raise from 25K to 30K



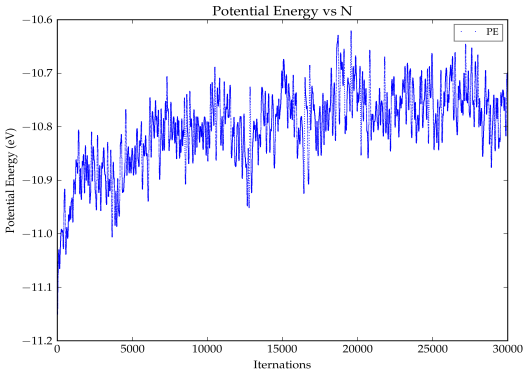
(a) Final Configuration and hydrostatic pressure



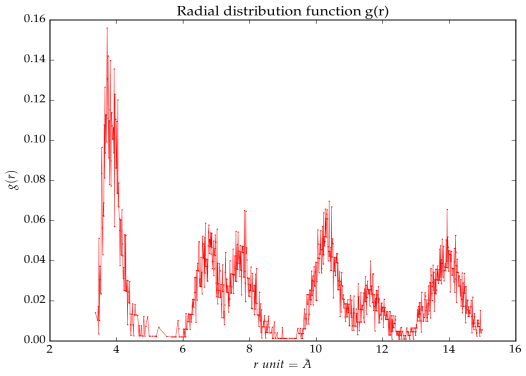
(b) Final Configuration



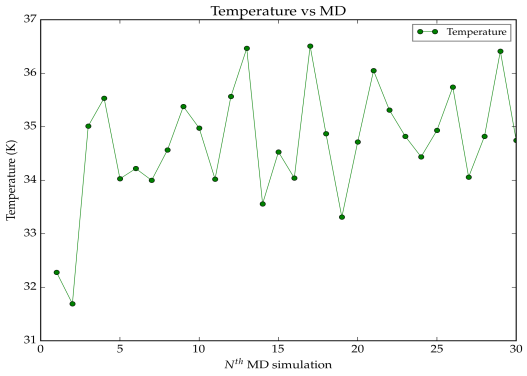
(c) Kinetic energy vs iteration



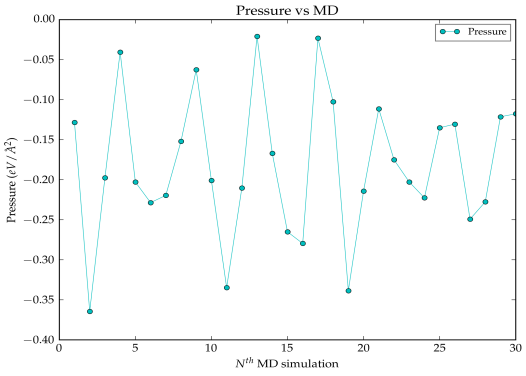
(d) Potential energy vs iteration



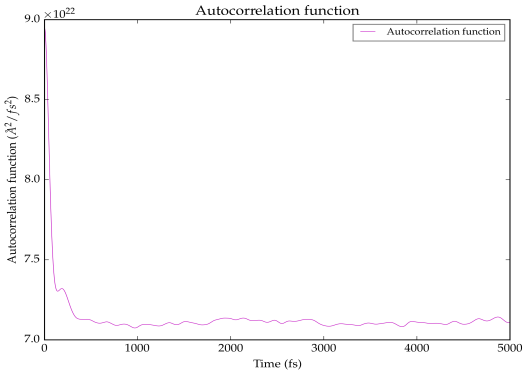
(e) Final Radial distribution function



(f) Temperature vs iteration

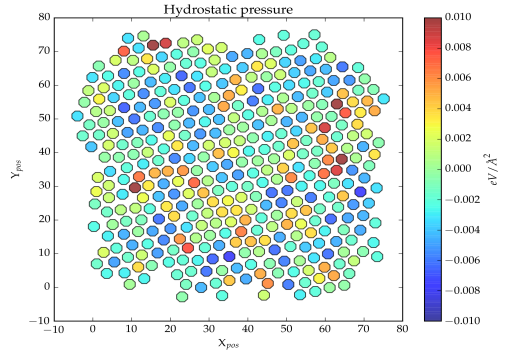


(g) Hydrostatic Pressure vs iteration

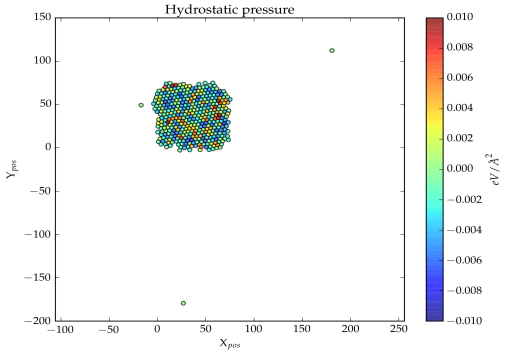


(h) Autocorrelation function

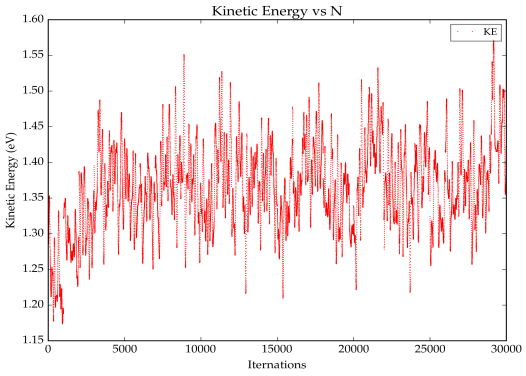
Figure 8: Temperature raise from 30K to 35K



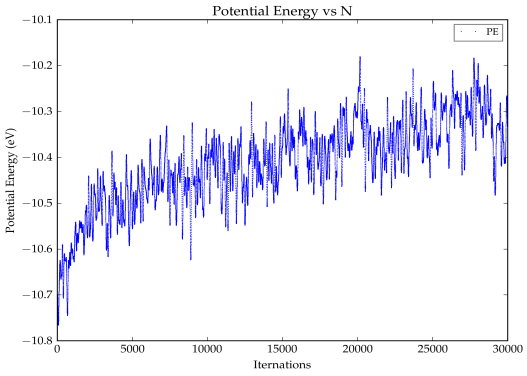
(a) Final Configuration and hydrostatic pressure



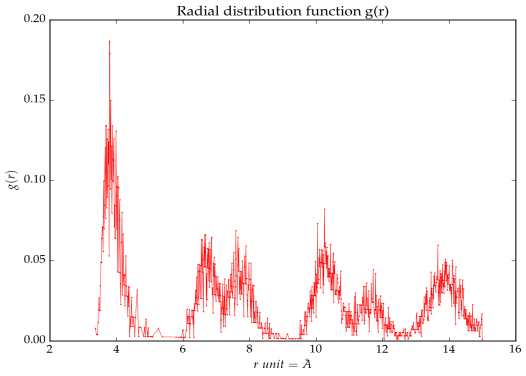
(b) Final Configuration



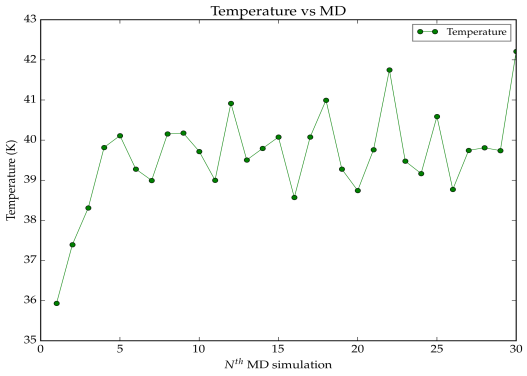
(c) Kinetic energy vs iteration



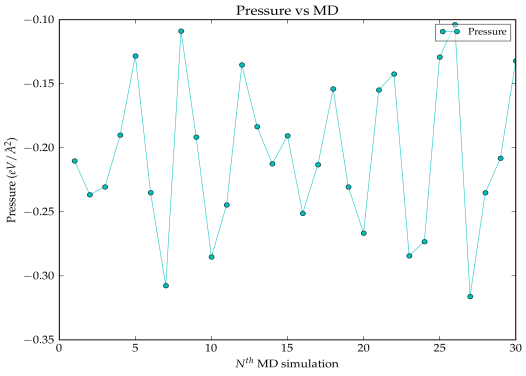
(d) Potential energy vs iteration



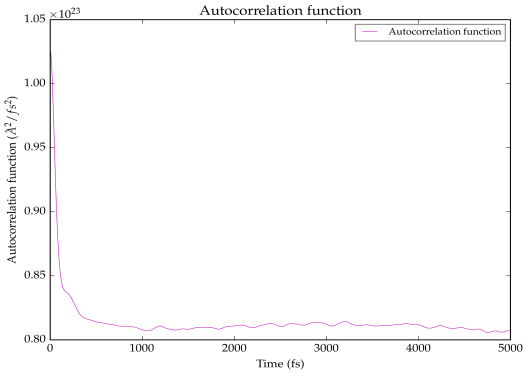
(e) Final Radial distribution function



(f) Temperature vs iteration

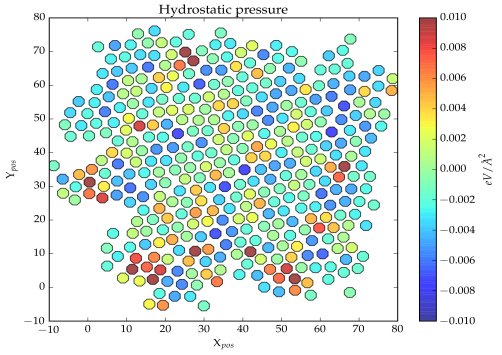


(g) Hydrostatic Pressure vs iteration

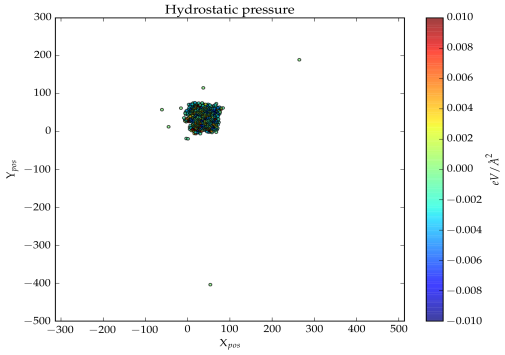


(h) Autocorrelation function

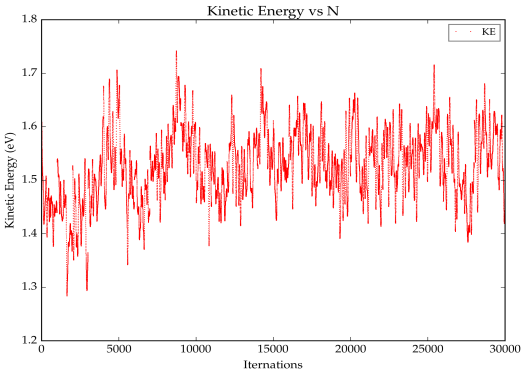
Figure 9: Temperature raise from 35K to 40K



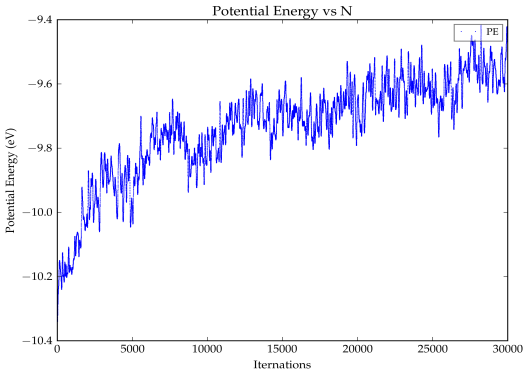
(a) Final Configuration and hydrostatic pressure



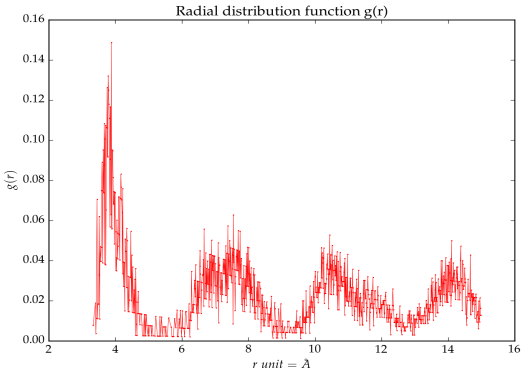
(b) Final Configuration



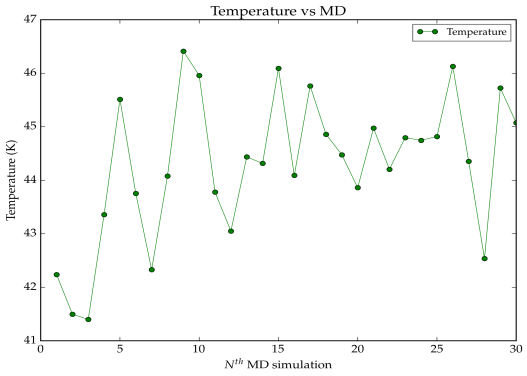
(c) Kinetic energy vs iteration



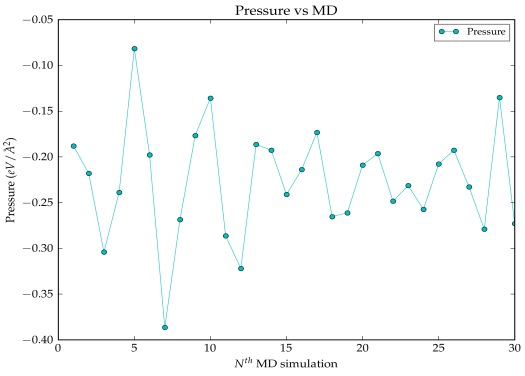
(d) Potential energy vs iteration



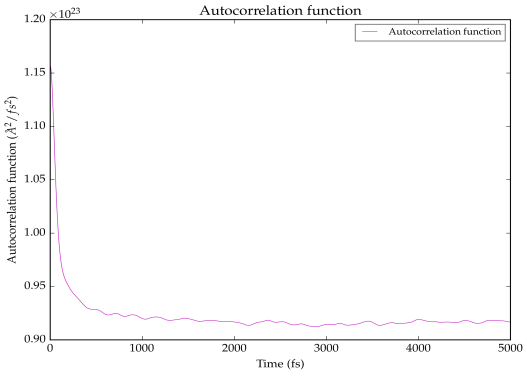
(e) Final Radial distribution function



(f) Temperature vs iteration

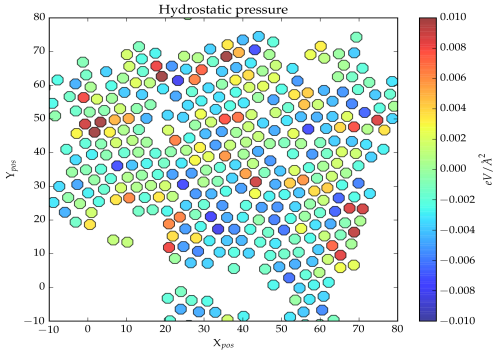


(g) Hydrostatic Pressure vs iteration

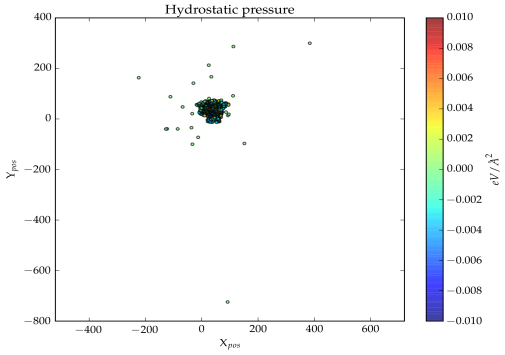


(h) Autocorrelation function

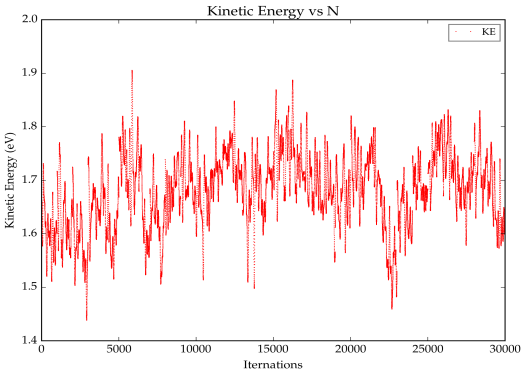
Figure 10: Temperature raise from 40K to 45K



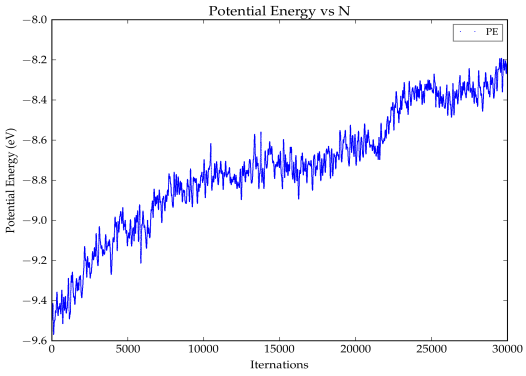
(a) Final Configuration and hydrostatic pressure



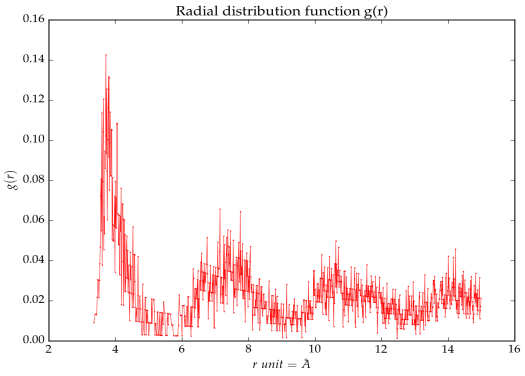
(b) Final Configuration



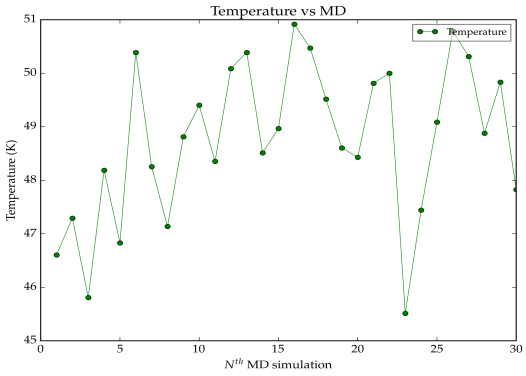
(c) Kinetic energy vs iteration



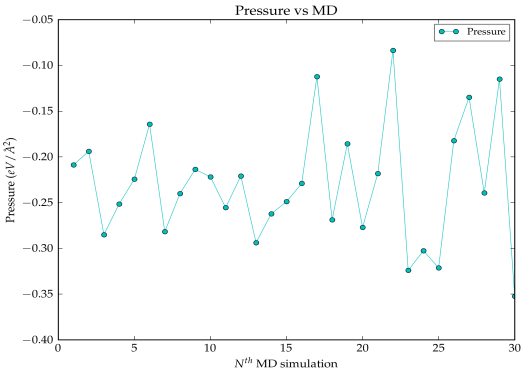
(d) Potential energy vs iteration



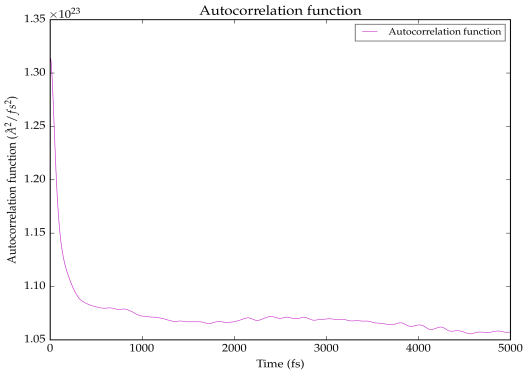
(e) Final Radial distribution function



(f) Temperature vs iteration

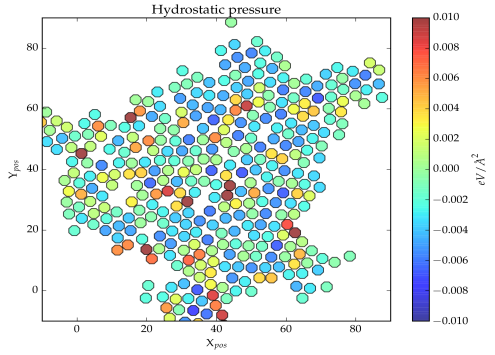


(g) Hydrostatic Pressure vs iteration

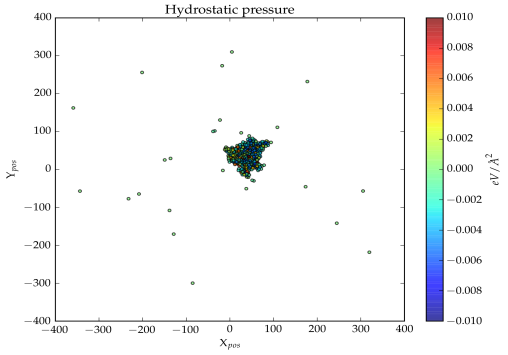


(h) Autocorrelation function

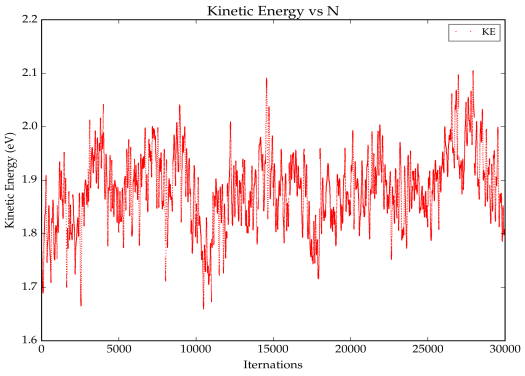
Figure 11: Temperature raise from 45K to 50K



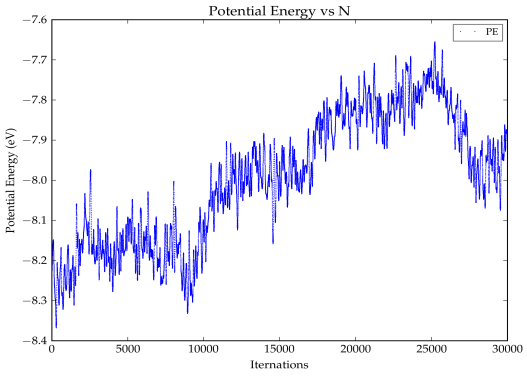
(a) Final Configuration and hydrostatic pressure



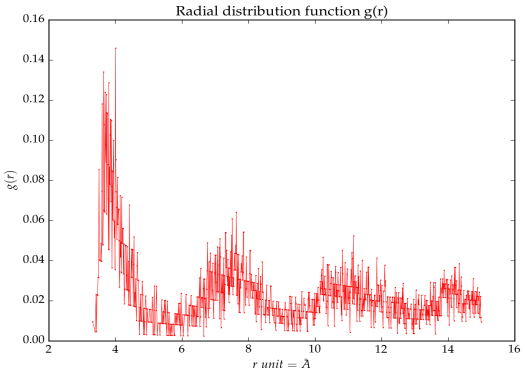
(b) Final Configuration



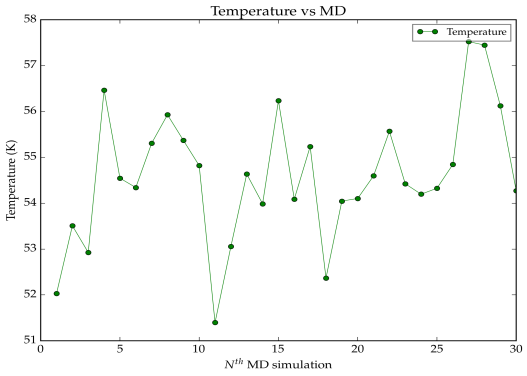
(c) Kinetic energy vs iteration



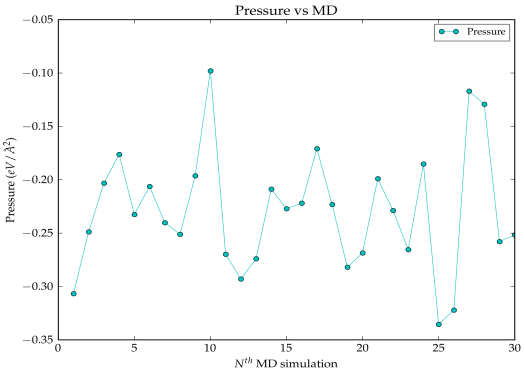
(d) Potential energy vs iteration



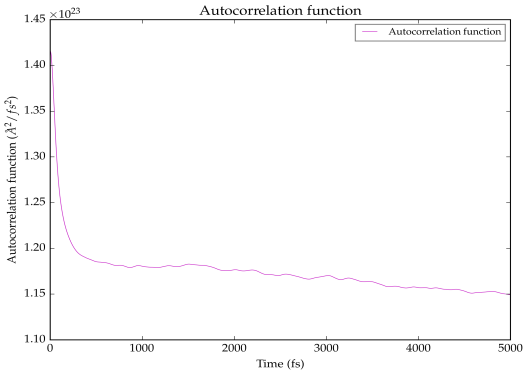
(e) Final Radial distribution function



(f) Temperature vs iteration

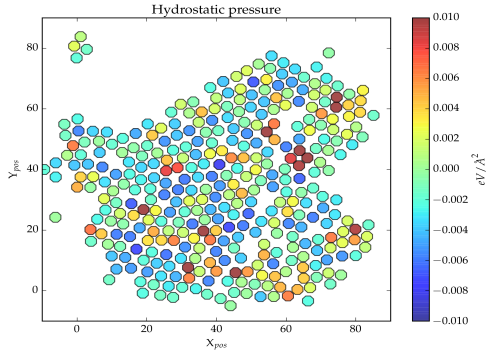


(g) Hydrostatic Pressure vs iteration

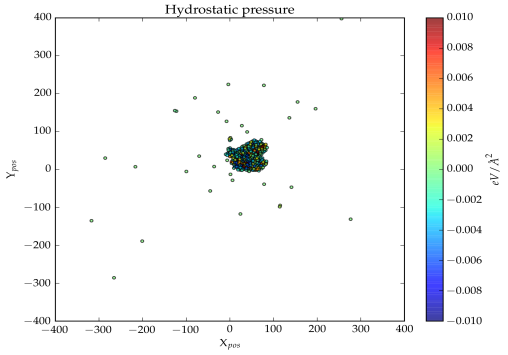


(h) Autocorrelation function

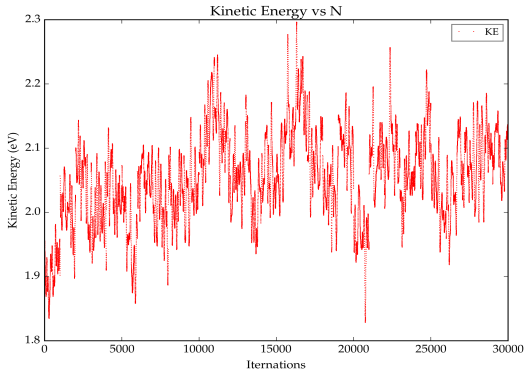
Figure 12: Temperature raise from 50K to 55K



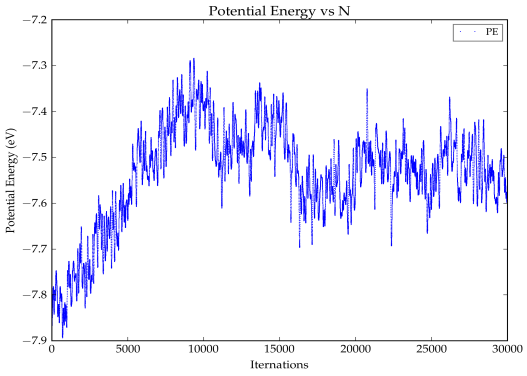
(a) Final Configuration and hydrostatic pressure



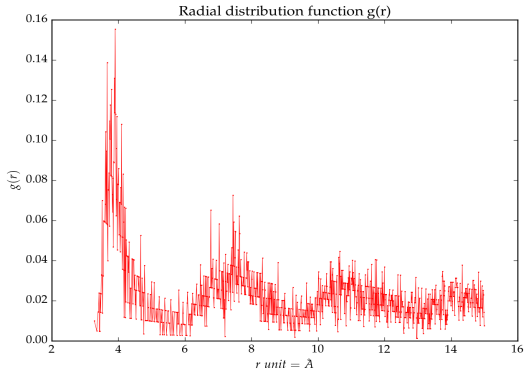
(b) Final Configuration



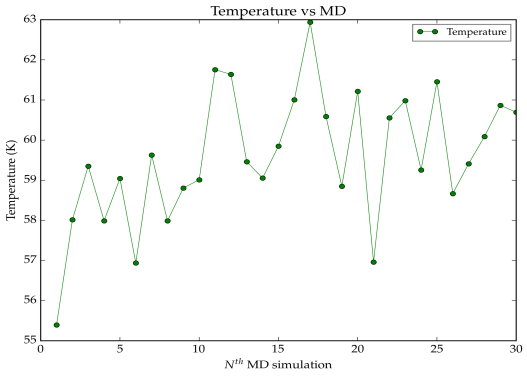
(c) Kinetic energy vs iteration



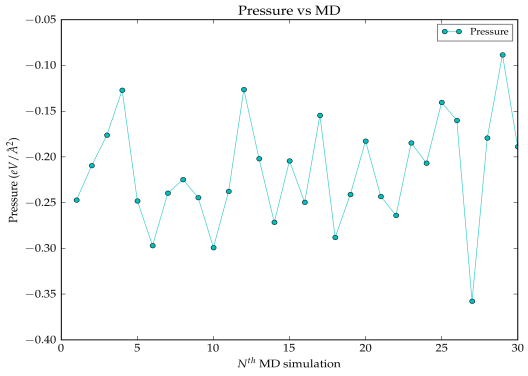
(d) Potential energy vs iteration



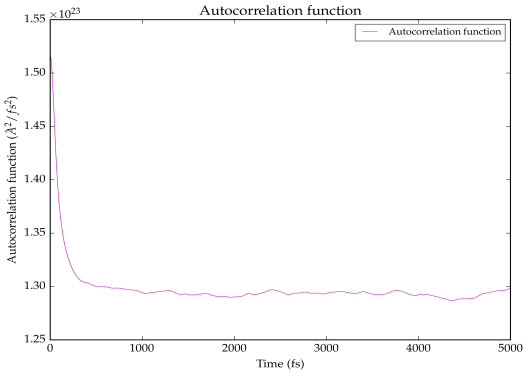
(e) Final Radial distribution function



(f) Temperature vs iteration

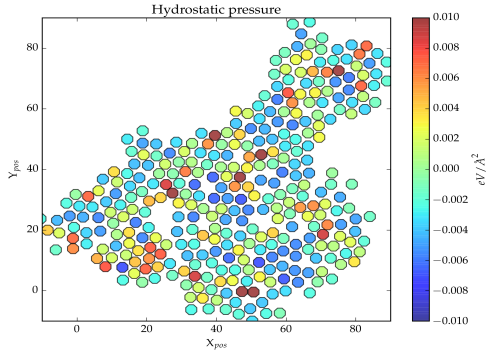


(g) Hydrostatic Pressure vs iteration

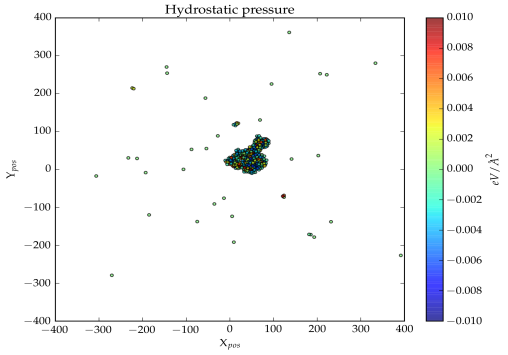


(h) Autocorrelation function

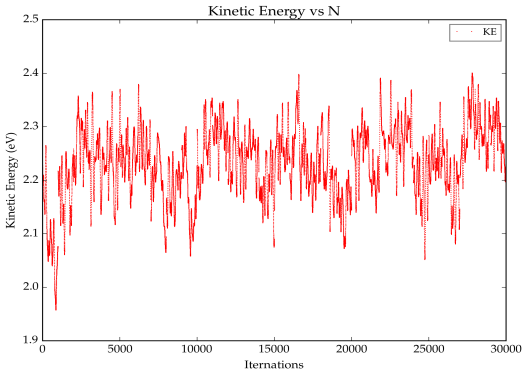
Figure 13: Temperature raise from 55K to 60K



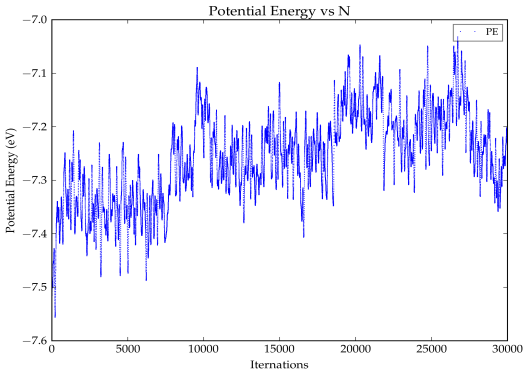
(a) Final Configuration and hydrostatic pressure



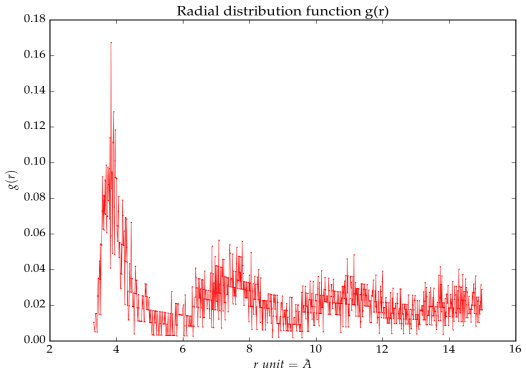
(b) Final Configuration



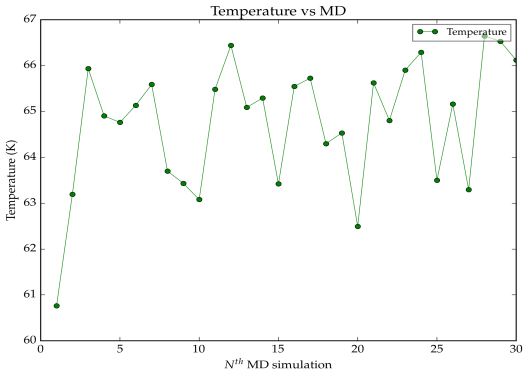
(c) Kinetic energy vs iteration



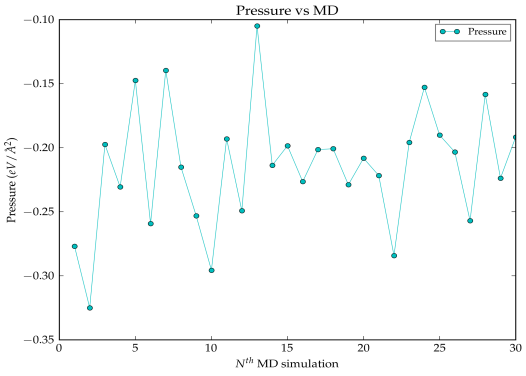
(d) Potential energy vs iteration



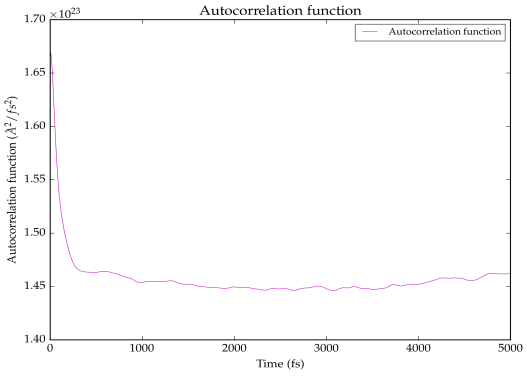
(e) Final Radial distribution function



(f) Temperature vs iteration

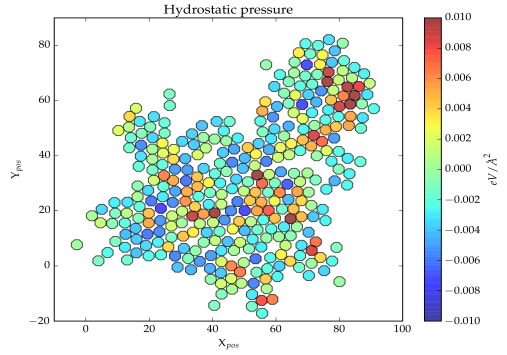


(g) Hydrostatic Pressure vs iteration

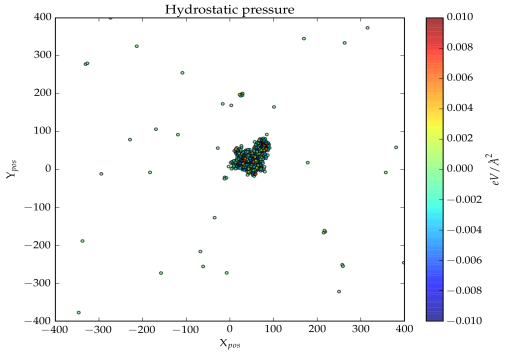


(h) Autocorrelation function

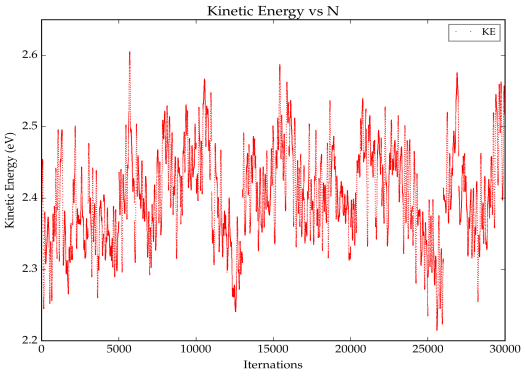
Figure 14: Temperature raise from 60K to 65K



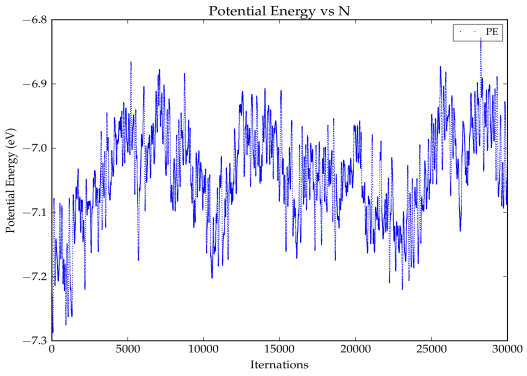
(a) Final Configuration and hydrostatic pressure



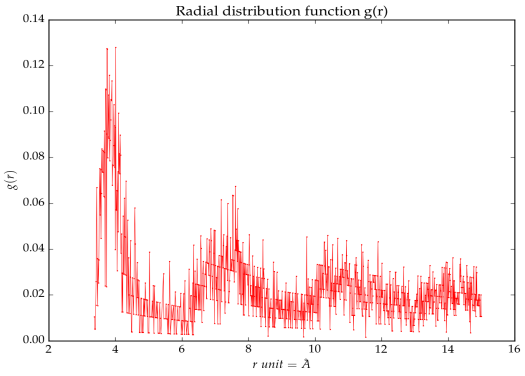
(b) Final Configuration



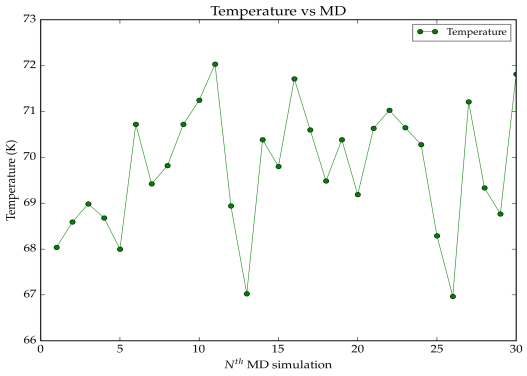
(c) Kinetic energy vs iteration



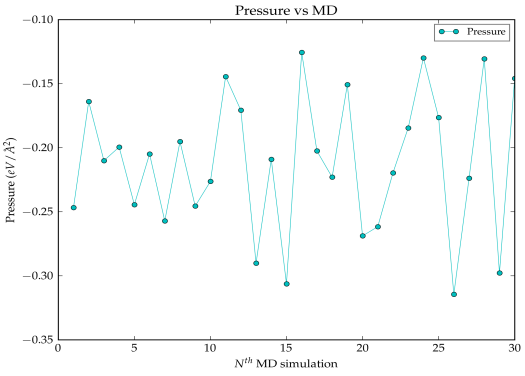
(d) Potential energy vs iteration



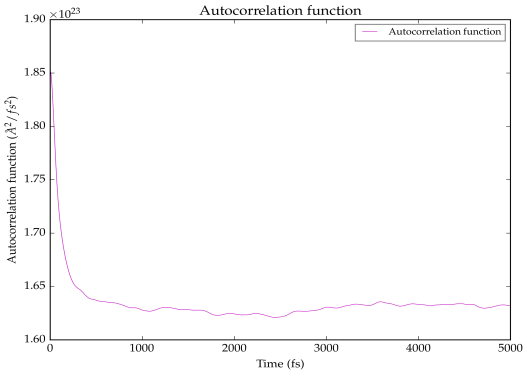
(e) Final Radial distribution function



(f) Temperature vs iteration

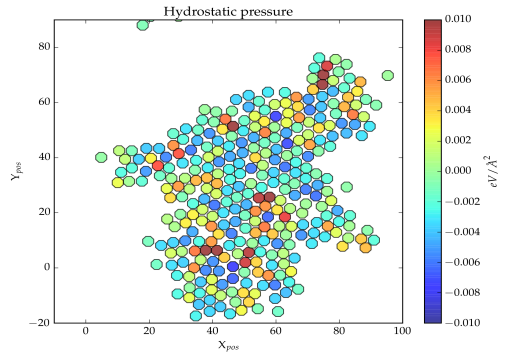


(g) Hydrostatic Pressure vs iteration

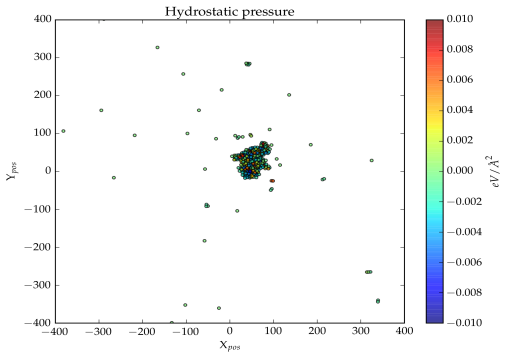


(h) Autocorrelation function

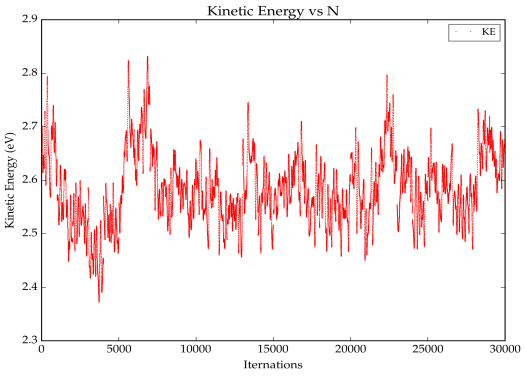
Figure 15: Temperature raise from 65K to 70K



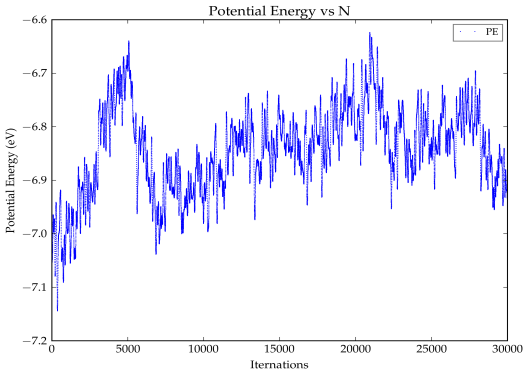
(a) Final Configuration and hydrostatic pressure



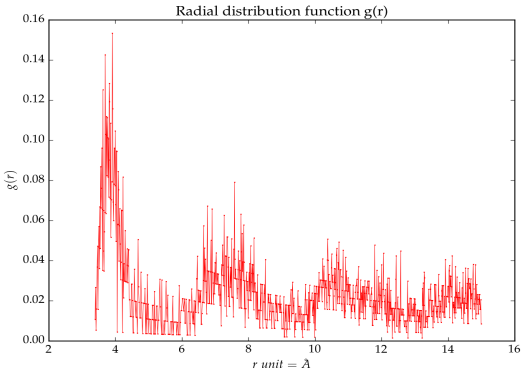
(b) Final Configuration



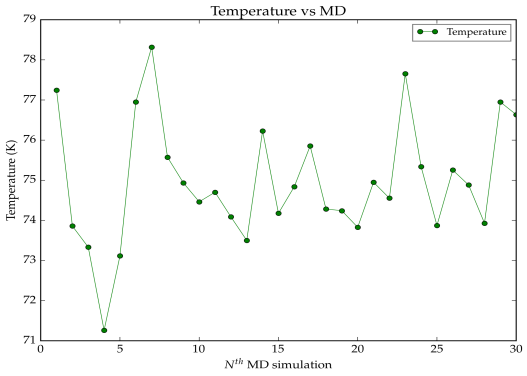
(c) Kinetic energy vs iteration



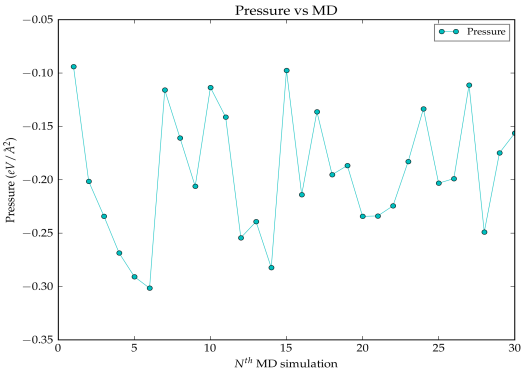
(d) Potential energy vs iteration



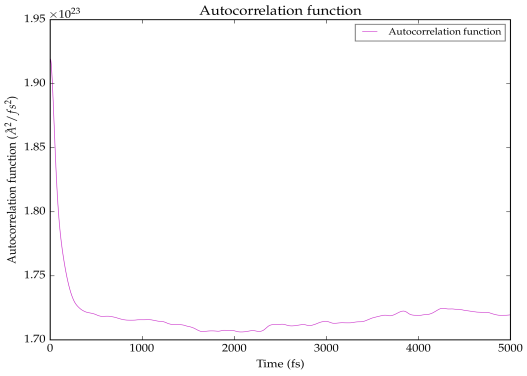
(e) Final Radial distribution function



(f) Temperature vs iteration

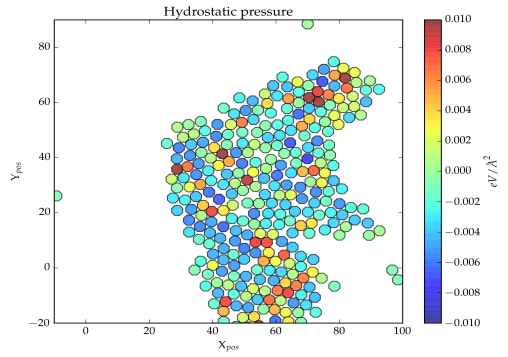


(g) Hydrostatic Pressure vs iteration

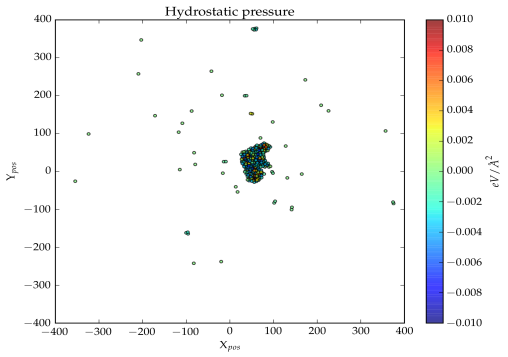


(h) Autocorrelation function

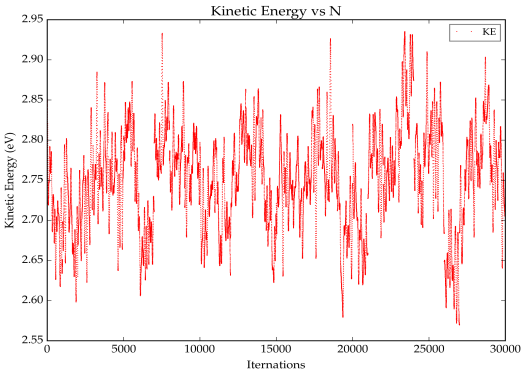
Figure 16: Temperature raise from 70K to 75K



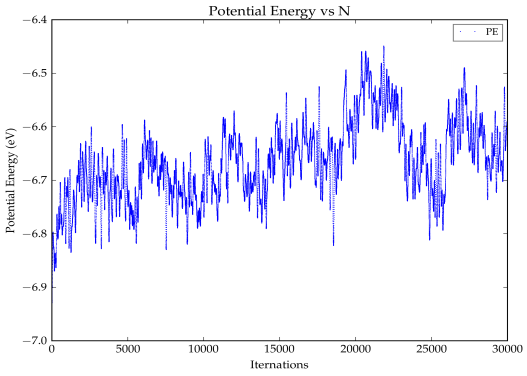
(a) Final Configuration and hydrostatic pressure



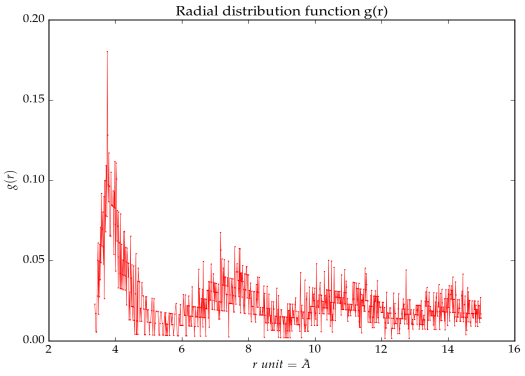
(b) Final Configuration



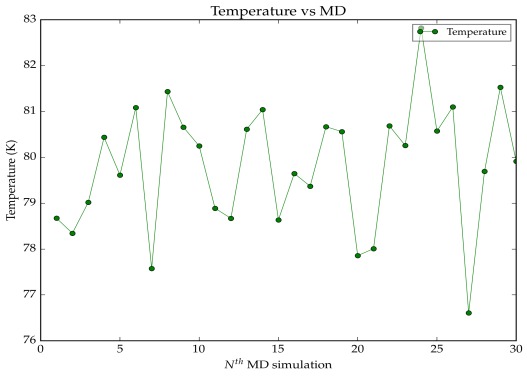
(c) Kinetic energy vs iteration



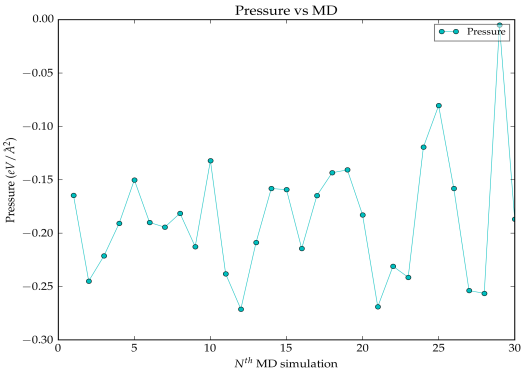
(d) Potential energy vs iteration



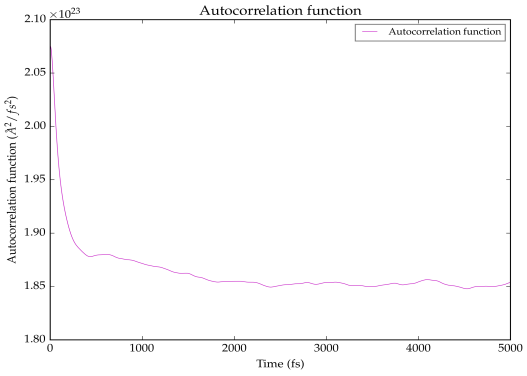
(e) Final Radial distribution function



(f) Temperature vs iteration

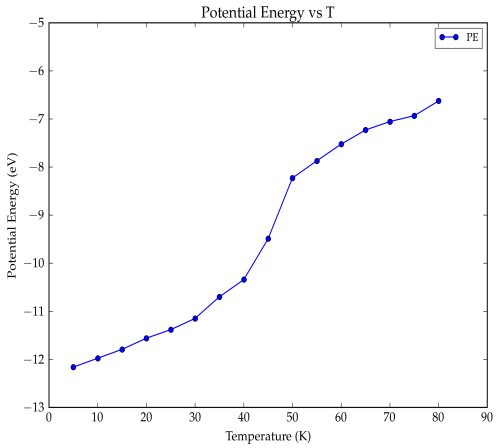


(g) Hydrostatic Pressure vs iteration

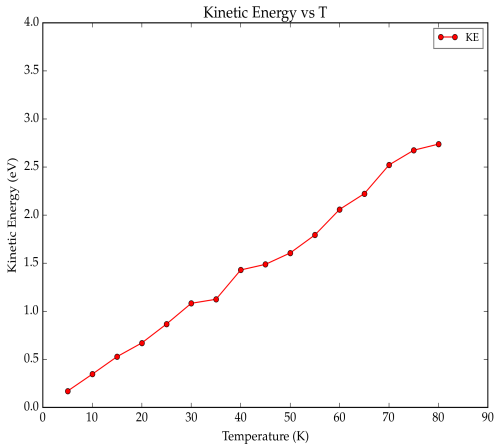


(h) Autocorrelation function

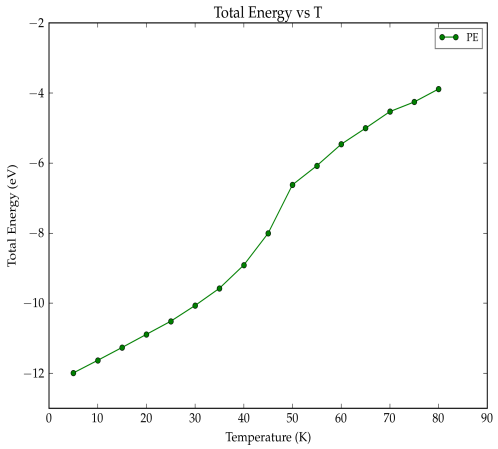
Figure 17: Temperature raise from 75K to 80K



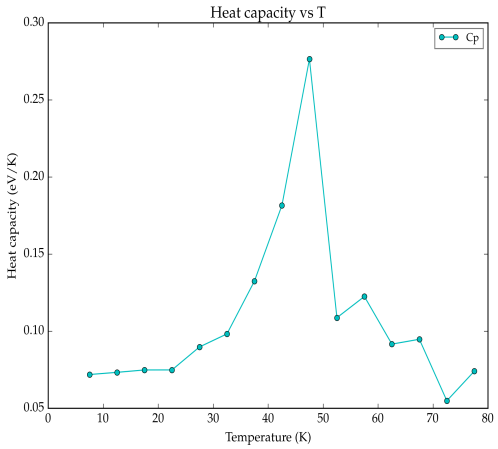
(a) Potential Energy vs Temperature



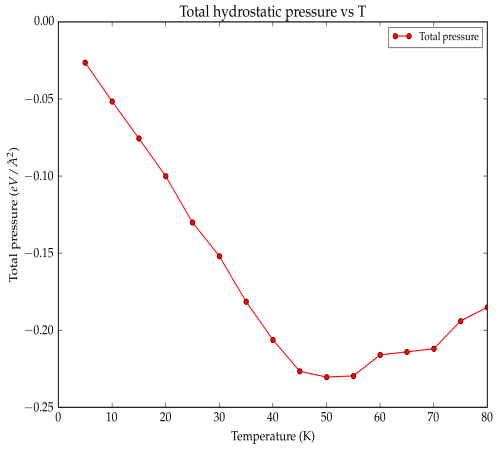
(b) Kinetic Energy vs Temperature



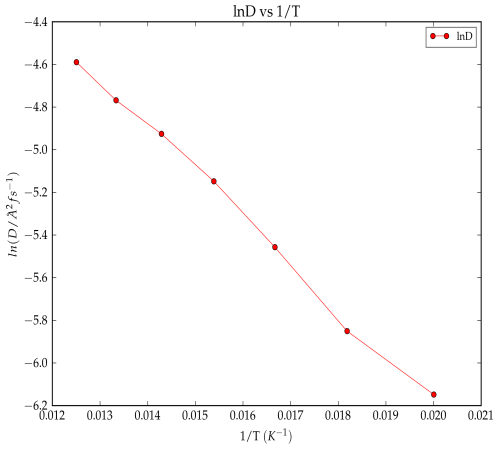
(c) Total Energy vs Temperature



(d) Heat Capacity vs Temperature



(e) Total pressure vs Temperature



(f) lnD vs 1/T

Figure 18: Energy,C_p, P_{tot} and self-diffusion coefficient at different Temperatures

3 Conclusion

1. Phase transformation start at around 40K as expected and took place from 40K to 60K. The signature of phase transformation: **I. a jump at E vs T plot** and **II. a peak at Heat capacity vs T plot** are both observed even though not as steep/sharp as the real case and this is due to the fact that our model of 400 atoms is too small compared to the real situation.

2. The Heat capacity, which has been calculated by its definition based on the plot of E vs T, shows a peak from 40K to 60K, which agree with our conclusion that phase transformation has been taken place from 40K to 60K.

3. 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 at 75K to 80K, possibly a gas phase has been formed such that the material might be a mixture of liquid and gas.

4. From 5K to 50K, almost linearly, the total hydrostatic pressure keep decreasing as the temperature increase. After the phase changed from solid to liquid, from 50K to 80K, the pressure starts to increase gradually.

5. The self-diffusion coefficient plot shows that lnD is linearly depend on 1/T, which agrees with the theory since $lnD = lnD_o + \frac{-\Delta E}{k_B T}$. According to our data, the slope $\frac{-\Delta E}{k_B} = -213.33K$, hence the corresponding activation energy $\Delta E = k_B * 213.33K = 0.018eV$

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
// HW4 Moleculardynamics
// 2-D Block
// Author: Yuding Ai
// Date: 2017.03.21

#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 3
 * Since now the atoms have velocities, I modified the structure
 * of each atom to be array<double,9> with a 8th and 9th component
 * to be v_x, and v_y. So now my atom looks like the following:
 *
 * Atom(xposition,yposition, sigmaxx, sigmayy, sigmaxy, F_x, F_y,v_x,v_y)
 *
 * 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 = 0.01;
const double neighCut = 7.5;
const double PI = 3.1415926;
```

```
47
const double k_b = 8.617*1E-5; //Boltzman's constant eV*K^-1
49 const double mass = 39.948*1.66E-27; // the mass of argon atom kg

51 // 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
53
//++++++++++ A shot comment on the choice of delta_t: ++++++++++++++++
55 //In class, we have learned that delta_t should be at least smaller than
//1E-13 and a proper choice would be between 1E-14 and 1E-15.
57 //
//I have tried simulations for both delta_t = 1E-14 and 1E-15 for the first
59 //part of this assignment to minimize the potential energy using Molecular
//dynamics. According to my simulations, it turns out the delta_t = 1E-14
61 //is a better choice over 1E-15 due to a faster computation and better
//results.(see details result in the report)
63 //
//Hence for the temperature dependence simulations, we set delta_t = 1E-14
65
class block
67 {
    private:
69      int w;          // Width;
        int h;          // Height;
71      int n;          // number of atoms;

73      // each atom is represented by an array
        array<array<double,9>,400> atomlist;
75      // Once the idx is known, all the info about idxth atom is known
        // because we could reach such atom by: block[idx]
77
        //a list of neighbor lists of each atom
79      array<vector<int>,400> neighbor_list;

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

89      ///-------------------------------------------------------------
        /// getter (subroutines)
91      ///-------------------------------------------------------------

93      int getN();

95      // 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]
97      // for example: its position is (block[idx][0],block[idx][1]) and its
        // stresses are (block[idx][2],block[idx][3],block[idx][4])
99      array<double,9> &operator[](int index);
```

```
101
        // return the distance between two atoms ----r_{ik}
103     double get_distance(int idx1, int idx2) const;

105     // return the distance in x axis between two atoms ----r_{ik}^{alpha}
        double get_xdistance(int idx1, int idx2) const;
107
        // return the distance in y axis between two atoms ----r_{ik}^{beta}
109     double get_ydistance(int idx1, int idx2) const;

111     //------------get_neighbor_list---------------------------------
        // take a single argument r, which is neighCut in this assignment
113     // and will update neighbor_list of current configuration.
        // Using Spencer's trick to reduce the computational complicity
115     // ------------------------------------------------------------
        void get_neighbor_list(double r);
117
        //------------get_atomic_stress---------------------------------
119     //take no argument and compute the atomic stresses of all the
        //atoms and assign it to each atom.
121     // ------------------------------------------------------------
        void get_atomic_stress();
123
        //------------rdf() ---------------------------------------------
125     //Get the radial distribution and output into a txt file
        // ------------------------------------------------------------
127     void rdf(string filename);


129
        //------------out_config()--------------------------------------
131     // output the configuration of atoms for latter plotting
        // and visualization
133     // ------------------------------------------------------------
        void out_config(string filename);
135
        //------------total_penergy()-----------------------------------
137     // compute the total potential energy of a configuration by sum over
        // the potential energy of all atoms
139     // E_tot = 1/2 sum_{i,j} psi(r_{i,j})
        // ------------------------------------------------------------
141     double total_penergy();

143     //------------calc_force() & calc_single_force()----------------
        //compute the force of a configuration for each atom by:
145     //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
147     //then find and return the maximum force;
        //
149     //calc_single_force()
        //compute the net force on a single atom and assign such
151     //force onto atomlist[idx][5] and atomlist[idx][6]
        //
153     //qcalc_mforce() calculate the maximum force a current atomlist
        //by simply sort the force component
```

```
155         // ------------------------------------------------------------
            double calc_force();
157         void calc_single_force(int idx);
            double qcalc_mforce();
159
161         //------------calc_total_stress_pressure()---------------------
            //compute the total stress sigma_xx, simga_yy and simga_xy and
163         //the hydrostatic pressure P. Store the 4 values into an array
            //such that (sum_simga_xx,sum_simga_yy,sum_simga_xy,P)
165         //------------------------------------------------------------
            array<double,4> calc_total_stress_pressure();
167
            //------------SD(double lambda)-------------------------------
169         //perform a steepest decent minimization
            //in my case, for each SD, perform 8000 iterations
171         // ------------------------------------------------------------
            void SD(double lambda,int it,string filename);
173
            //===============Methods for HW 4================================
175
            /*----------- set_config(filename) ----------------------------
177          * manully set the configuration for the block,so including
             * position, velocities, Atomic Force and so on (9 value per atom)
179          */
            void set_config(string filename);
181
            /*-----------update_position(int idx)--------------------------
183          * update the position of each atom using verlet algorithm
             * then assign the velocity into each atom
185          */
            void update_position();
187
            /*-----------update_velosity(int idx)--------------------------
189          * update the velocity of each atom using verlet algorithm
             * then assign the velocity into each atom
191          */
            void update_velosity(array<double,400> F_oldx, array<double,400>
    F_oldy);
193
            /* ---------- Molecular Dynamics MD_mini() : HW part1 -----------
195          * perform a molecular dynamics simulation using verlet algorithm
             * verlet algorithm  to minimize the potential energy
197          */
            void MD_mini(int it);
199

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

207         /* ---------- autocorrelation() --------------------------------
```

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

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

        double diffusion(int it);

};

/// 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
// HW4 Molecular dynamics
// 2-D Block
// Author: Yuding Ai
// Date: 2017.03.21

#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;
    vector<double> dislist;
    vector<double> Rlist;
    vector<double> glist;
    get_neighbor_list(rdfMax);

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

    //dislist is sorted in increment order
```

```
    sort(dislist.begin(),dislist.end());

    double d = dislist[0];

    // count each distance
    int c = 0;
    for (unsigned int i = 0; i < dislist.size(); i++) {
        if(abs(dislist[i] - d)<= deltaR){
            c++;
        }
        else{
            //rlist is a list of possible distance of all neighbors
            //sorted in increment order
            Rlist.push_back(d);
            //glist is a list of occurrence/frequency of ith r for now
            glist.push_back(1.0*c/dislist.size()); //divide dislist to
    normalize it
            c = 0;
            d = dislist[i];
        }

        if(i == dislist.size()-1){
            //rlist is a list of possible distance of all neighbors
            //sorted in increment order
            Rlist.push_back(d);
            //glist is a list of occurrence/frequency of ith r for now
            glist.push_back(1.0*c/dislist.size()); //divide dislist.size() to
    normalize it
            c = 0;
            d = dislist[i];
        }
    }

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

    // record rdf into a txt file for latter plotting
    ofstream myfile(filename);
    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;
}
```

```
298 void block::SD(double lambda,int it,string filename){
        stringstream st;
300     int j = 0;
        int k = 0;
302     while(j<it){
            for (int i = 0; i < n; i++) {
304             // get the force on each atom
                calc_single_force(i);
306             // Move each atom following the direction
                // of it's force
308             atomlist[i][0] += lambda*atomlist[i][5];
                atomlist[i][1] += lambda*atomlist[i][6];
310         }

312         if(it>=1000){
                if(k==it/1000){
314                 //first update the neighborlist
                    get_neighbor_list(neighCut);
316                 //update the stresses
                    get_atomic_stress();
318                 //calc hydrostatic pressure
                    array<double,4> sp = calc_total_stress_pressure();
320                 //calc total potential energy
                    double E = total_penergy();
322                 //calc maximum force
                    double F = qcalc_mforce();
324                 //record the state/configuration data
                    st<<j<<" "<<E<<" "<<sp[0]<<" "<<sp[1]<<" ";
326                 st<<sp[2]<<" "<<sp[3]<<" "<<F<<endl;
                    k = 0;
328             }
            }
330         else{
                //first update the neighborlist
332             get_neighbor_list(neighCut);
                //update the stresses
334             get_atomic_stress();
                //calc hydrostatic pressure
336             array<double,4> sp = calc_total_stress_pressure();
                //calc total energy
338             double E = total_penergy();
                //calc maximum force
340             double F = qcalc_mforce();
                //record the state/configuration data
342             st<<j<<" "<<E<<" "<<sp[0]<<" "<<sp[1]<<" ";
                st<<sp[2]<<" "<<sp[3]<<" "<<F<<endl;
344         }

346         j++;
            k++;
348     }
        ofstream myfile(filename);
350     string data = st.str();
```

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

void block::set_config(string filename){
    string line;
    ifstream myfile(filename);
    if(myfile.is_open()){
        int i = 0;
        while (std::getline(myfile,line)){
            stringstream linestream(line);
            string data;
            std::getline(linestream,data,' ');
            atomlist[i][0] = std::stod(data);
            linestream >>atomlist[i][1]>>atomlist[i][2]>>
                atomlist[i][3]>>atomlist[i][4]
                >>atomlist[i][5]>>atomlist[i][6]
                >>atomlist[i][7]>>atomlist[i][8];
            i++;
        }
    }
}
void block::update_position(){
    /* According to Verlet algorithm
     * r_i((j+1)*deltat) = r_i(j*deltat) + v_i(j*deltat)*deltat +
     * (deltat)^2/2m_i *F_i(j*deltat)
     */

    //Recall in my atom's data structure,
    //atom[0]  --- xposition
    //atom[1]  --- yposition
    //atom[5]  --- F_x
    //atom[6]  --- F_y
    //atom[7]  --- v_x
    //atom[8]  --- v_y
    //And when we calc r, F and v are all corresponding to j*delte_t
    for(unsigned int idx = 0; idx< atomlist.size();idx++){
        //x_position
        atomlist[idx][0] = atomlist[idx][0] + atomlist[idx][7]*delta_t
            + delta_t*delta_t/(2*mass)* atomlist[idx][5];

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

void block::update_velosity(array<double,400> F_oldx, array<double,400> F_oldy
    ){
    /* According to Verlet algorithm
     * v_i((j+1)*deltat) = v_i(j*deltat) + deltat/2m_i*(F_i((j+1)*deltat)
     * + F_i(j*deltat) )
     *
     * To have F_i((j+1)*delta_t), we should first store the old Force
```

```
404      * F_i(j*delta_t) then update the force corresponding to (j+1)*delta_t
         * configuration
406      */
     for(unsigned int idx = 0; idx< atomlist.size();idx++){
408      //v_x
         atomlist[idx][7] = atomlist[idx][7] + delta_t/(2*mass) *(atomlist[idx
    ][5]+ F_oldx[idx]);
410
         //v_y
412      atomlist[idx][8] = atomlist[idx][8] + delta_t/(2*mass) *(atomlist[idx
    ][6]+ F_oldy[idx]);
     }
414 }

416 void block::MD_mini(int it){
     stringstream st;
418  stringstream st_temp_stress;
     array<double,4> totstress; //store the total stress and pressure
420  array<double,10> pressurearray; //store the pressure of the 10 MD
    simulations
     array<double,10> temperaturearray; //store the temperature of the 10 MD
    simulations
422  array<double,400> F_oldx;
     array<double,400> F_oldy;
424
     int counter = 0;
426  //I have tried many runs on this model and
     //according to my experiments:
428  //when delta_t = 1E-14:
     //A. At least 30000 iterations is needed for the first MD
430  //simulation to reach the equalibirum. For the next few MD simulations
     //, fewer and fewer steps is needed to reach its equalibirum. so
432  //here we choose 50000 iterations for each MD simulation.
     //and after 30000 iterations, we start to collect date as to calculate
434  //the equalibirum average.
     //
436  //B. I found that 10 times of sucessive MD simulation with 50000
     //iterations for each MD simulation would be sufficient enough
438  //to reach the minimun potential energy.(PE_min = -12.3343 eV) which is
     //even better than the result from my previous Molecular static method
440  //(-11.7813 eV)
     //
442  //Therefore, here we do 10 successive MD simulations to minimize
     //the potential energy.
444
     for (int n = 0; n<10;n++){
446      //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++){
448          atomlist[j][7] = 0;
             atomlist[j][8] = 0;
450      }
         double KE= 0;
452      double KEsum = 0;
         double Psum=0;
```

```
454
        //then get the initial Force and potential energy
456     double PE = total_penergy();
        calc_force();
458
        int i = 0;
460     while(i < it){
            for(unsigned int j = 0; j< atomlist.size();j++){
462             F_oldx[j] = atomlist[j][5];
                F_oldy[j] = atomlist[j][6];
464         }
            //for eacn iteration, first get the new position
466         update_position();
            //then once atoms are moved, we update the neighbor_list
468         get_neighbor_list(neighCut);
            // calc the new force
470         calc_force();
            //then get the new velocity(the new force is been updated during
472         update_velosity(F_oldx, F_oldy);

474         //calc the PE and KE corresponding to the new configuration
            PE = total_penergy();
476         KE = 0; //recalc the KE
            for (unsigned int k = 0; k<atomlist.size();k++){
478             KE += 0.5* mass*(atomlist[k][7]*atomlist[k][7] +
                        atomlist[k][8]*atomlist[k][8]);
480         }
            //store the infomation into stringtream
482         st<< KE <<" "<<PE<<" "<<counter<<endl;
            cout<< KE <<" "<<PE<<" "<<counter<<endl;
484         i++;
            counter++;
486
            //According to my result, for delta_t = 1E-14, equalibirum would
    reached
488         //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>
490         //FYI, My program takes: 4213.73 sec to compele 10 sets of MD
    simulation
            //with 50000 itetarions each. so the computation is pretty slow
492
            if(i>30000){
494             KEsum +=KE;
                totstress = calc_total_stress_pressure();
496             Psum += totstress[3];
            }
498     }

500     KEsum /=(it - 30000); //get the <KE>
        Psum /=(it - 30000); //get the <P>
502     double temp = KEsum/(400*k_b);
        get_atomic_stress();
```

```
504         totstress = calc_total_stress_pressure();
            pressurearray[n] = totstress[3];
506         temperaturearray[n] = temp;

508         st_temp_stress<<temp <<" "<<totstress[0]<<" "<<totstress[1]<<" "<<
        totstress[2]
                <<" "<<totstress[3]<<" "<<Psum<<endl;
510     }

512     ofstream myfile1("MD_temp_stress.txt");
        string data1 = st_temp_stress.str();
514     myfile1<< data1;
        myfile1.close();
516
        ofstream myfile("MD_Energy.txt");
518     string data = st.str();
        myfile<< data;
520     myfile.close();
}
522

524 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
526     // but this time is to simulate the temperature dependence MD
        stringstream st;
528     stringstream st_temp_stress;
        array<double,4> totstress; //store the total stress and pressure
530     array<double,400> F_oldx;
        array<double,400> F_oldy;
532
        int counter = 0;
534
        //before starts the MD simulation, we first calculate k_factor based on
        the
536     //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
538     //
        //as K = (2Nk_b*T_desire/sum(m_i*<vi^2>))^0.5 = (T_desire/T_old)^0.5
540
        double k_fac = pow(desire_T/old_T,0.5); //set initial k_fac
542     // for (int n = 0; n<20;n++){
        for (int n = 0; n<30;n++){
544         //For each MD, before looping, we first scale the velocIty
            for(unsigned int j = 0; j< atomlist.size();j++){
546             atomlist[j][7] = atomlist[j][7]*k_fac;
                atomlist[j][8] = atomlist[j][8]*k_fac;
548         }
            double KE= 0;
550         double velocity = 0;
            double KEsum_tempdenpendent = 0;
552         double hydroP=0;
            double Tdependent_temp = 0;
554
```

```
        //then get the initial Force and potential energy
556     double PE = total_penergy();
        calc_force();
558
        int i = 0;
560     while(i < it){
            //get the old force
562         for(unsigned int j = 0; j< atomlist.size();j++){
                F_oldx[j] = atomlist[j][5];
564             F_oldy[j] = atomlist[j][6];
            }
566         //for eacn iteration, first get the new position
            update_position();
568         //then once atoms are moved, we update the neighbor_list
            get_neighbor_list(neighCut);
570         // calc the new force
            calc_force();
572         //then get the new velocity(the new force is been updated during
            update_velosity(F_oldx, F_oldy);
574
            //calc the PE and KE corresponding to the new configuration
576         PE = total_penergy();
            KE = 0; //recalc the KE
578         for (unsigned int k = 0; k<atomlist.size();k++){
                KE += 0.5* mass*(atomlist[k][7]*atomlist[k][7] +
580                     atomlist[k][8]*atomlist[k][8]);
                velocity += pow(atomlist[k][7]*atomlist[k][7]+
582                     atomlist[k][8]*atomlist[k][8],0.5);

584         }

586         get_atomic_stress();
            totstress = calc_total_stress_pressure();
588         velocity /=atomlist.size();
            cout<< KE <<" "<<PE<<" "<<" "<<velocity<<" "<<counter<<" "<<
    totstress[3]<<endl;
590         st<< KE <<" "<<PE<<" "<<" "<<velocity<<" "<<counter<<" "<<
    totstress[3]<<endl;
            i++;
592         counter++;
            KEsum_tempdenpendent +=KE;
594         hydroP +=totstress[3];

596     }

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

606     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;

    double velocity = 0;
    while(i < it){
        velocity = 0;
        //get the old force
        for(unsigned int j = 0; j< atomlist.size();j++){
            F_oldx[j] = atomlist[j][5];
            F_oldy[j] = atomlist[j][6];
        }
        //for eacn iteration, first get the new position
        update_position();
        //then once atoms are moved, we update the neighbor_list
        get_neighbor_list(neighCut);
        // calc the new force
        calc_force();
        //then get the new velocity(the new force is been updated during
```

```
        update_velosity(F_oldx, F_oldy);

        //calc the PE and KE corresponding to the new configuration
        PE = total_penergy();
        KE = 0; //recalc the KE
        array<double,400> v_run;
        for (unsigned int k = 0; k<atomlist.size();k++){
            KE += 0.5* mass*(atomlist[k][7]*atomlist[k][7] +
                    atomlist[k][8]*atomlist[k][8]);

            velocity = pow(atomlist[k][7]*atomlist[k][7]+
                    atomlist[k][8]*atomlist[k][8],0.5);
            v_run[k] = velocity;
        }
        v_all.push_back(v_run);
        cout << counter<<"velocity "<<KE <<" "<<velocity<<endl;

        i++;
        counter++;
        KEsum_tempdenpendent +=KE;
    }

    double autoco = 0;
    double vo = 0;
    double vt = 0;
    for (int k = 0; k<5000;k++){
        autoco = 0;
        for(int i = 0; i<5000;i++ ){
            for(int j = 0; j<400;j++){
                vo = v_all[i][j];
                vt = v_all[i+k][j];
                autoco = autoco + vo*vt;
            }
        }
        autoco = autoco/(400*5000);
        st<<autoco<<" "<<k<<endl;
    }

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

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

    int counter = 0;
```

```
716     double KE= 0;
        double KEsum_tempdenpendent = 0;
718
        //then get the initial Force and potential energy
720     double PE = total_penergy();
        calc_force();
722
        int i = 0;
724     counter = 0;

726     double velocity = 0;
        while(i < it){
728         velocity = 0;
            //get the old force
730         for(unsigned int j = 0; j< atomlist.size();j++){
                F_oldx[j] = atomlist[j][5];
732             F_oldy[j] = atomlist[j][6];
            }
734         //for eacn iteration, first get the new position
            update_position();
736         //then once atoms are moved, we update the neighbor_list
            get_neighbor_list(neighCut);
738         // calc the new force
            calc_force();
740         //then get the new velocity(the new force is been updated during
            update_velosity(F_oldx, F_oldy);
742
            //calc the PE and KE corresponding to the new configuration
744         PE = total_penergy();
            KE = 0; //recalc the KE
746         array<double,400> r_runx;
            array<double,400> r_runy;
748         for (unsigned int k = 0; k<atomlist.size();k++){
                KE += 0.5* mass*(atomlist[k][7]*atomlist[k][7] +
750                     atomlist[k][8]*atomlist[k][8]);

752             r_runx[k] = atomlist[k][0];
                r_runy[k] = atomlist[k][1];
754         }
            r_x.push_back(r_runx);
756         r_y.push_back(r_runy);
            cout << counter<<"velocity "<<KE <<" "<<velocity<<endl;
758
            i++;
760         counter++;
            KEsum_tempdenpendent +=KE;
762     }

764     double rox = 0;
        double roy = 0;
766     double rtx = 0;
        double rty = 0;
768     double D= 0;
```

```
    for (int k = 0; k<5000;k++){
        D = 0;
        for(int i = 0; i<5000;i++ ){
            for(int j = 0; j<400;j++){
                rox = r_x[i][j];
                roy = r_y[i][j];
                rtx = r_x[i+k][j];
                rty = r_y[i+k][j];
                D += (rtx-rox)*(rtx-rox)+(rty-roy)*(rty-roy);
            }
        }
        D = D/(400*5000);
    }
    D /= (6*5000);
    double lnD=0;
    lnD = log(D);
    return lnD;
}

///------------------------------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
// HW4 Molecular Dynamics
// Main function
// Author: Yuding Ai
// Penn ID: 31295008
// Data: 2017.03.21

#include "2dblock.h"

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

```
13
    //-------------------------------------------------
15  // Minimize the Potential energy by MD
    //-------------------------------------------------
17  // bloc1.MD_mini(100000); //for delta_t = 1e-15
    bloc1.MD_mini(50000); //for delta_t = 1e-14
19  bloc1.out_config("MD_minimization.txt");
    bloc1.rdf("rdf_MDmini.txt");
21
    // //-------------------------------------------------
23  // // Simulate the temperature dependence by MD
    // //-------------------------------------------------
25  // //-------- T_0 = 5K----------------------------------
    // bloc1.set_config("mini/MD_minimization.txt");
27  // bloc1.MD(1000,5,0.0921569,"5K");
    // bloc1.out_config("5K/MD_minimization.txt");
29  // bloc1.rdf("5K/rdf_MDmini.txt");
    //
31  // //-------- T_0 = 10K---------------------------------
    // bloc1.set_config("5K/MD_minimization.txt");
33  // bloc1.MD(1000,10,5,"10K");
    // bloc1.out_config("10K/MD_minimization.txt");
35  // bloc1.rdf("10K/rdf_MDmini.txt");
    //
37  // //-------- T_0 = 15K---------------------------------
    // bloc1.set_config("10K/MD_minimization.txt");
39  // bloc1.MD(1000,15,10,"15K");
    //
41  // bloc1.out_config("15K/MD_minimization.txt");
    // bloc1.rdf("15K/rdf_MDmini.txt");
43  //
    // //-------- T_0 = 20K---------------------------------
45  // bloc1.set_config("15K/MD_minimization.txt");
    // bloc1.MD(1000,20,15,"20K");
47  //
    // bloc1.out_config("20K/MD_minimization.txt");
49  // bloc1.rdf("20K/rdf_MDmini.txt");
    //
51  // //-------- T_0 = 25K---------------------------------
    // bloc1.set_config("20K/MD_minimization.txt");
53  // bloc1.MD(1000,25,20,"25K");
    //
55  // bloc1.out_config("25K/MD_minimization.txt");
    // bloc1.rdf("25K/rdf_MDmini.txt");
57  //
    // //-------- T_0 = 30K---------------------------------
59  // bloc1.set_config("25K/MD_minimization.txt");
    // bloc1.MD(1000,30,25,"30K");
61  //
    // bloc1.out_config("30K/MD_minimization.txt");
63  // bloc1.rdf("30K/rdf_MDmini.txt");
    //
65  // //from now on, the phase is starting to change
    //
```

```
67      // //-------- T_0 = 35K---------------------------------
        // bloc1.set_config("30K/MD_minimization.txt");
69      // bloc1.MD(1000,35,30,"35K");
        //
71      // bloc1.out_config("35K/MD_minimization.txt");
        // bloc1.rdf("35K/rdf_MDmini.txt");
73      //
        // //-------- T_0 = 40K---------------------------------
75      // bloc1.set_config("35K/MD_minimization.txt");
        // bloc1.MD(1000,40,35,"40K");
77      //
        // bloc1.out_config("40K/MD_minimization.txt");
79      // bloc1.rdf("40/rdf_MDmini.txt");
        //
81      // // //-------- T_0 = 45K---------------------------------
        // bloc1.set_config("40K/MD_minimization.txt");
83      // bloc1.MD(1000,45,40,"45K");
        //
85      // bloc1.out_config("45K/MD_minimization.txt");
        // bloc1.rdf("45K/rdf_MDmini.txt");
87      //
        // //-------- T_0 = 50K---------------------------------
89      // bloc1.set_config("45K/MD_minimization.txt");
        // bloc1.MD(1000,50,45,"50K");
91      //
        // bloc1.out_config("50K/MD_minimization.txt");
93      // bloc1.rdf("50K/rdf_MDmini.txt");
        //
95      // //-------- T_0 = 55K---------------------------------
        // bloc1.set_config("50K/MD_minimization.txt");
97      // bloc1.MD(1000,55,50,"55K");
        //
99      // bloc1.out_config("55K/MD_minimization.txt");
        // bloc1.rdf("55K/rdf_MDmini.txt");
101     // //-------- T_0 = 60K---------------------------------
        //
103     // bloc1.set_config("55K/MD_minimization.txt");
        // bloc1.MD(1000,60,55,"60K");
105     //
        // bloc1.out_config("60K/MD_minimization.txt");
107     // bloc1.rdf("60K/rdf_MDmini.txt");
        //
109     // // //-------- T_0 = 65K---------------------------------
        // bloc1.set_config("60K/MD_minimization.txt");
111     // bloc1.MD(1000,65,60,"65K");
        //
113     // bloc1.out_config("65K/MD_minimization.txt");
        // bloc1.rdf("65K/rdf_MDmini.txt");
115     //
        // // //-------- T_0 = 70K---------------------------------
117     // bloc1.set_config("65K/MD_minimization.txt");
        // bloc1.MD(1000,70,65,"70K");
119     //
        // bloc1.out_config("70K/MD_minimization.txt");
```

```
121     // bloc1.rdf("70K/rdf_MDmini.txt");
        //
123     // // //-------- T_0 = 75K----------------------------------
        //  bloc1.set_config("70K/MD_minimization.txt");
125     // bloc1.MD(1000,75,70,"75K");
        //
127     // bloc1.out_config("75K/MD_minimization.txt");
        // bloc1.rdf("75K/rdf_MDmini.txt");
129     //
        // //-------- T_0 = 80K----------------------------------
131     // bloc1.set_config("75K/MD_minimization.txt");
        // bloc1.MD(1000,80,75,"80K");
133     //
        // bloc1.out_config("80K/MD_minimization.txt");
135     // bloc1.rdf("80K/rdf_MDmini.txt");
        //
137     // //---------------auto correlation--------------------------
        // // takes 4345 secs to run auto correlation
139     // // ------------------------------------------------------
        // bloc1.set_config("5K/MD_minimization.txt");
141     // bloc1.autocorrelation(20000,"5K");
        //
143     // bloc1.set_config("10K/MD_minimization.txt");
        // bloc1.autocorrelation(20000,"10K");
145     //
        // bloc1.set_config("15K/MD_minimization.txt");
147     // bloc1.autocorrelation(20000,"15K");
        //
149     // bloc1.set_config("20K/MD_minimization.txt");
        // bloc1.autocorrelation(20000,"20K");
151     //
        // bloc1.set_config("25K/MD_minimization.txt");
153     // bloc1.autocorrelation(20000,"25K");
        //
155     // bloc1.set_config("30K/MD_minimization.txt");
        // bloc1.autocorrelation(20000,"30K");
157     //
        // bloc1.set_config("35K/MD_minimization.txt");
159     // bloc1.autocorrelation(20000,"35K");
        //
161     // bloc1.set_config("40K/MD_minimization.txt");
        // bloc1.autocorrelation(20000,"40K");
163     //
        // bloc1.set_config("45K/MD_minimization.txt");
165     // bloc1.autocorrelation(20000,"45K");
        //
167     // bloc1.set_config("50K/MD_minimization.txt");
        // bloc1.autocorrelation(20000,"50K");
169     //
        // bloc1.set_config("55K/MD_minimization.txt");
171     // bloc1.autocorrelation(20000,"55K");
        //
173     // bloc1.set_config("60K/MD_minimization.txt");
        // bloc1.autocorrelation(20000,"60K");
```

```
    //
    // bloc1.set_config("65K/MD_minimization.txt");
    // bloc1.autocorrelation(20000,"65K");
    //
    // bloc1.set_config("70K/MD_minimization.txt");
    // bloc1.autocorrelation(20000,"70K");
    //
    // bloc1.set_config("75K/MD_minimization.txt");
    // bloc1.autocorrelation(20000,"75K");
    //
    // bloc1.set_config("80K/MD_minimization.txt");
    // bloc1.autocorrelation(20000,"80K");
    //
    // // ------------------diffusion ----------------
    // stringstream s;
    // bloc1.set_config("50K/MD_minimization.txt");
    // s<<1.0/50<<" "<<bloc1.diffusion(20000)<<endl;
    //
    // bloc1.set_config("55K/MD_minimization.txt");
    // s<<1.0/55<<" "<<bloc1.diffusion(20000)<<endl;
    //
    // bloc1.set_config("60K/MD_minimization.txt");
    // s<<1.0/60<<" "<<bloc1.diffusion(20000)<<endl;
    //
    // bloc1.set_config("65K/MD_minimization.txt");
    // s<<1.0/65<<" "<<bloc1.diffusion(20000)<<endl;
    //
    // bloc1.set_config("70K/MD_minimization.txt");
    // s<<1.0/70<<" "<<bloc1.diffusion(20000)<<endl;
    //
    // bloc1.set_config("75K/MD_minimization.txt");
    // s<<1.0/75<<" "<<bloc1.diffusion(20000)<<endl;
    //
    // bloc1.set_config("80K/MD_minimization.txt");
    // s<<1.0/80<<" "<<bloc1.diffusion(20000)<<endl;
    //
    // string filename = "selfdiffusion_v2.txt";
    // ofstream myfile(filename);
    // string data = s.str();
    // myfile<< data;
    // myfile.close();
/*  */

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

Listing 3: main.cpp

```
# Diffusion.py
# HW4 Molecular Dynamics
# Author: Yuding Ai
# Date: 2017.03.21
```

```
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 itertools import groupby
rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)

def diffuse():
    lnD = []
    T_rev = [];
    with open("selfdiffusion_v2.txt","r") as file:
        for line in file:
            words = line.split()
            trev = float(words[0])
            lnd = float(words[1])
            lnD.append(lnd)
            T_rev.append(trev)


    fig = plt.figure()
    plt.plot(T_rev,lnD,'ro-',linewidth = 0.5,markersize = 5,label = r'lnD')
    leg = plt.legend(prop={'size':10})
    leg.get_frame().set_alpha(0.5)
    xlabel = r'1/T $(K^{-1})$'
    ylabel = r'$ln(D/\AA^2fs^{-1})$'
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.xlim([0.012,0.021])
    plt.title(r'lnD vs 1/T')
    fig.savefig("selfDiffuse_v2",dpi = 300, bbox_inches ='tight')


def main():
    diffuse()

main()
```

Listing 4: Diffusion.py

```
# hw4.py
# HW4 Molecular Dynamics
# Author: Yuding Ai
# Date: 2017.03.21

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 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([-4,90])
    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"$eV/{\AA}^2$")
    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 = "hydrop" + filename
    titlename = r"Hydrostatic pressure"
    plt.title(titlename)
    fig.savefig(fname,dpi=300)



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()

    if txtname == "rdf.txt":
        plt.stem(R, G, linefmt='b-', linewidth = 0.4,markerfmt='rx', basefmt='
   r-',label = r'RDF')
    else:
        plt.plot(R,G,'rx-',linewidth = 0.4,markersize = 1,label = r'RDF')
    xlabel = r'$r\ unit =\AA $'
    ylabel = r'$g(r)\ $'
    plt.title('Radial distribution function g(r)')
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    fig.savefig(filename, dpi=180, bbox_inches='tight')

def E_vs_it():
    KE = [] # a list of KE
    PE = [] # a list of PE
    IT = [] # Iteration
    with open("part2_MD_Energy.txt","r") as file:
        for line in file:
            words = line.split()
            ke = float(words[0]) #take the value
            pe = float(words[1]) #take the value
            it = float(words[3]) #take the value

            KE.append(ke);
            PE.append(pe);
            IT.append(it);

    fig = plt.figure()
    plt.plot(IT,KE,'r+',linewidth = 0.1,markersize = 1,label = r'KE')
    leg = plt.legend(prop={'size':10})
    leg.get_frame().set_alpha(0.5)
    xlabel = r'Iternations'
```

```
    ylabel = r'Kinetic Energy (eV)'
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(r'Kinetic Energy vs N')
    fig.savefig("KEvsIT",dpi = 300, bbox_inches ='tight')


    fig = plt.figure()
    plt.plot(IT,PE,'b+',linewidth = 0.1,markersize = 1,label = r'PE')
    leg = plt.legend(prop={'size':10})
    leg.get_frame().set_alpha(0.5)
    xlabel = r'Iternations'
    ylabel = r'Potential Energy (eV)'
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(r'Potential Energy vs N')
    fig.savefig("PEvsIT",dpi = 300, bbox_inches ='tight')

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

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

    avep = sum(P)/len(P)
    print avep

    fig = plt.figure()
    plt.plot(step,T,'go-',linewidth = 0.5,markersize = 5,label = r'Temperature
    ')
    leg = plt.legend(prop={'size':10})
    leg.get_frame().set_alpha(0.5)
    xlabel = r'$N^{th}$ MD simulation'
    ylabel = r'Temperature (K)'
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(r'Temperature vs MD')
    fig.savefig("temperature",dpi = 300, bbox_inches ='tight')

    fig = plt.figure()
    plt.plot(step,P,'co-',linewidth = 0.5,markersize = 5,label = r'Pressure')
    leg = plt.legend(prop={'size':10})
    leg.get_frame().set_alpha(0.5)
    xlabel = r'$N^{th}$ MD simulation'
```

```
    ylabel = r'Pressure ($eV/\AA^2$)'
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(r'Pressure vs MD')
    fig.savefig("Pressure",dpi = 300, bbox_inches ='tight')



def main():
    E_vs_it()
    plotconfig("MD_minimization.txt","config.png")
    plotrdf("rdf_MDmini.txt","rdf.png")
    plottemp_pressure("part2_MD_temp_stress.txt")

main()
```

Listing 5: hw4.py

```
# autocor.py
# HW4 Molecular Dynamics
# Author: Yuding Ai
# Date: 2017.03.21

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 itertools import groupby
rc('font',**{'family':'serif','serif':['Palatino']})
rc('text', usetex=True)

def autocor():
    Auto = []
    T = []
    with open("auto.txt","r") as file:
        count = 0
        for line in file:
            words = line.split()
            a = float(words[0])
            t = float(words[1])
            Auto.append(a)
            T.append(t)

    fig = plt.figure()
    plt.plot(T,Auto,'m-',linewidth = 0.5,markersize = 5,label = r'
   Autocorrelation function')
    leg = plt.legend(prop={'size':10})
    leg.get_frame().set_alpha(0.5)
    xlabel = r'Time (fs)'
    ylabel = r'Autocorrelation function ($\AA^2/fs^2$)'
```

```
36      plt.xlabel(xlabel)
        plt.ylabel(ylabel)
38      plt.title(r'Autocorrelation function')
        fig.savefig("Auto",dpi = 300, bbox_inches ='tight')
40

42  def main():
        autocor()
44

    main()
```

Listing 6: autocor.py