Homework 3

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

March 12, 2017

1 Steepest descent Algorithm

In this work, we will employ the so-called steepest decent algorithm, which has been regarded as the simplest algorithm to minimize the potential energy, to perform a simulation on the 2d block of atoms from assignment 2.
Steepest decent (SD) algorithm take the 1st order approximation for the Energy change corresponding to a small displacement Δr, such that ΔE could be described as in Equation 1.

$$\Delta E_p = \sum_{\alpha=1}^{3} \frac{\partial E_p}{\partial r_i^\alpha} \Delta r_i^\alpha \tag{1}$$

SD algorithm also assume that the direction of the small displacement Δr of each atom would be the same as the net force acting on such atom (hence the "steepest decent"), where the net force are coming from its neighbors. The net force of i^{th} atom on m^{th} iteration could be described as in Equation 2.

$$F_i^m = -grad_{r_i} E_p(r_1^{m-1}, r_2^{m-1}, ..., r_N^{m-1}) \tag{2}$$

Once F is known, we could take

$$\Delta r_i^m = \lambda F_i^m \tag{3}$$

to move each atom by Δr_i^m, where λ is a numerical factor artificially chosen to optimize the procedure of minimize the total potential energy. (noticed λ can't be too big, see $\lambda = 8$ example in result section)

We will use the same Lenard Jones potential (Equation 4) and set $r_{tail} = 7.0\mathring{A}$ and $r_{cut} = 7.5\mathring{A}$ as well as all other constants (see details in Appendix: 2dblock.h, Constents) as we did in previous assignment.

$$\Phi(r) = \begin{cases} 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^6] & \text{if} \quad r \le r_o \\ A(r - r_{cut})^3 + B(r - r_{cut})^2 & \text{if} \quad r_o < r < r_{cut} \\ 0 & \text{if} \quad r \ge r_{cut} \end{cases} \tag{4}$$

Once Φ is known, the force on atom i at α direction could then be written as

$$F_i^\alpha = -\sum_{i,k} \frac{d\Phi(r_{ik})}{dr_{ik}} \frac{r_{ik}^\alpha}{r_{ik}} \tag{5}$$

where atom 'k' is a neighbor of atom 'i'. Similarly, the atomic stress could be calculate through Equation 6

$$\sigma_{\alpha\beta}^i = \begin{cases} \frac{1}{\Omega_i} \sum_{\substack{k \\ k \neq i}} \frac{24\epsilon}{r_{ik}} [-2(\frac{\sigma}{r_{ik}})^{12} + (\frac{\sigma}{r_{ik}})^6] \frac{r_{ik}^\alpha r_{ik}^\beta}{r_{ik}} & \text{if} \quad r \leq 7.0\mathring{A} \\ \frac{1}{\Omega_i} \sum_{\substack{k \\ k \neq i}} [3A(r_{ik} - r_{cut})^2 + 2B(r_{ik} - r_{cut})] \frac{r_{ik}^\alpha r_{ik}^\beta}{r_{ik}} & \text{if} \quad r > 7.0\mathring{A} \end{cases} \tag{6}$$

where $\Omega_i = r_{Min}^2(w-1)(h-1)/n$ is the average 'volume' (area in 2d) for each atom.

Armed with the above equations, we could then write a pseudo code for our simulation program as the following (see details implementation in Appendix, 2dblock.cpp, method block::SD())

Algorithm 1 Steepest Descent Algorithm

procedure STEEPEST DECENT METHOD
Set up the initial 2d block with 400 atoms
N iterations(start the loop)
    calculate the **Force on each atom** and find the **Maximum Force:** F_{max}
    Move each atom following the direction of its **force** using Equation 3
    Update the neighbor list (since we moved the atoms in previous step)
    calculate the **Potential Energy: E** of the current configuration
    calculate the **net Stress** $\sigma_{xx}, \sigma_{yy}, \sigma_{xy}$ and the **hydrostatic pressure: P**
Plot the **final configuration**
Calculate the **RDF** for the final configuration
Plot **P,** F_{max}**,** σ**, E VS Iterations**

2 Result

2.1 Compare different lambdas

First of all, as is mentioned in class, λ can't be too small (takes too long for the relaxation) or too big otherwise the 2d block will get stuck/blow up and thus atoms stop moving and failed to reach the equilibrium/minimum potential energy. Let's see what will happen with λ is too big. **Let $\lambda = 8$** (a too big value for λ) and Figure 1 illustrates the final configuration after 8000 iteration of Steepest Descent simulation. As is shown in Figure 1, we see that both the maximum force and potential energy blows up at several iterations and as a result, atoms moves too far away from the block and left with no force acting on it anymore. Finally, the maximum force became 0 when the block is far from equilibrium (final potential is about -5 eV). Therefore, λ can't be too big.

In this work, we choose λ **to be 0.5, 1, 1.5 and 2** which are all in a 'reasonable' (not too big) scale and it turns out that $\lambda = 1.5$ yields the best minimization result while $\lambda = 0.5$ fails to reach its relaxation state under 8000 iteration. For each λ value, we run a Steepest descent simulation with **8000 iterations** separately and independently to minimize the potential energy. (so 4 simulations in total) Utilizing the data obtained from those simulations, we plotted Figure 3, Figure 4, Figure 6 and Figure 7.

Figure 3 describes the **final configuration** of the equilibrium state (with minimum potential energy) for the 4 λ values and if we make comparison to the initial configuration,(see Figure 2) all 4 configurations have became more compact (similar to the closed packed structure) which is as expected due to a smaller total potential energy. Therefore it seems that all 4 simulations have reached a reasonable final state.

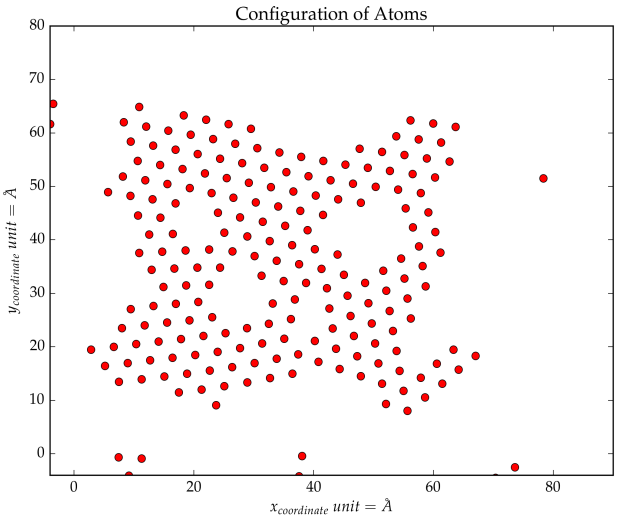
Figure 4 illustrate the **final Radial distribution function** for the 4 simulation, and we see that when λ is small (=0.5 for example), the peaks are fuzzy with lots of noise while as λ become bigger and bigger, the peaks become more and more clear and sharp. According to this result, $\lambda = 2$ and $\lambda = 1.5$ seems to have a more stable final configuration than $\lambda = 0.5$ and $\lambda = 1$. Further, if we count the peaks, we see that the $\lambda = 2$ and $\lambda = 1.5$ have a clear 7 peak distribution. Compare this RDF to the RDF for diamond in Figure 5[1], the two RDF are very similar thus indicates we have obtained a close packed structure (hexagonal structure in 2D).

Figure 6 depicts **potential energy E vs Iteration** and **maximum force F_{max} vs Iteration**, where the red, green, blue and cyan stands for $\lambda = 0.5$, 1, 1.5 and 2 respectively. Let's first look at the E vs Iteration plot, where the cyan line has the fastest rate to approach it's local minimum E and blue, green line has the 2nd and 3rd fast rate while the red line is the slowest one to approach it's local minimum. So that the higher the λ value, the faster for the simulation to approach the local minimum E. From the E vs Iteration plot, we could also see that the **blue line ($\lambda = 1.5$) reaches the lowest potential energy** $E_{1.5} = -11.7813eV$, the cyan line ($\lambda = 2$) obtained the second lowest energy $E_2 = -11.7461eV$ and the green line gets to $E_1 = -11.6322eV$, the red line gets to $E_{0.5} = -11.5612eV$. Even though, the cyan line ($\lambda = 2$) is leading through the first 1000 iterations among all the other lines, eventually, the blue line ($\lambda = 1.5$) take over the lead and reached the most stable state. From this, we could conclude that $\lambda = 1.5$ **yields the best minimization result among the 4 simulations**. Furthermore, in the F vs Iteration plot, we could also see that, even though at the beginning, all lines are fluctuated (and seems kind of messed up), at the end, the same sequence has appeared that blue line reaches the lowest maximum force and then the cyan, green and red line. Therefore, this agree with our conclusion from E vs Iteration plot.

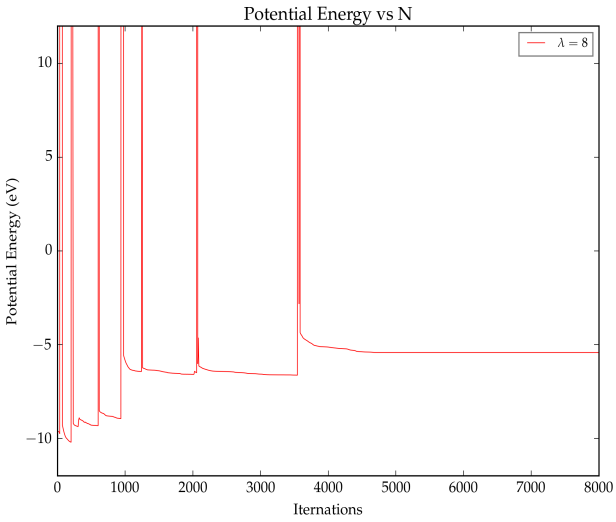
Last but not least, Figure 7 describes **Iteration vs the stresses $\sigma_{xx}, \sigma_{yy}, \sigma_{xy}$ and the hydrostatic pressure P** which is obtained by $P = -1/2 \times (\sigma_{xx} + \sigma_{yy})$. Once again, the red, green, blue and cyan stands for $\lambda = 0.5$, 1, 1.5 and 2 respectively. From the σ_{xx} vs Iteration plot, we see from most to least close to 0 σ_{xx} value, the sequence are cyan, blue, green and

[1]picture taken from https://arxiv.org/pdf/1112.5204.pdf

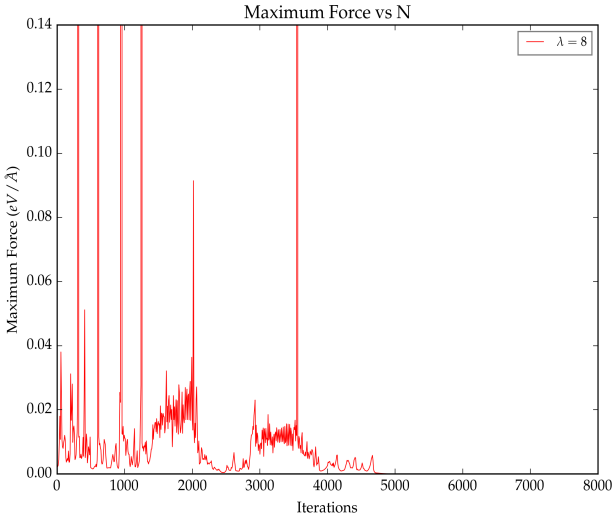
red, while from the σ_{yy} vs Iteration plot, such sequence switched to green, blue, cyan and red. However, when considering how stable the structure are, the hydrostatic pressure turns out to be more valuable than both σ_{xx} and σ_{yy} since it shows a relative average among σ_{xx} and σ_{yy}. According to the P vs Iteration plot, it shows that the blue, cyan and green all merge together to reach 0 pressure while the red is still apart from 0. This reflects that when λ is 2, 1.5 and 1, the block of atoms are getting to equilibrium (in the sense of stress) while when λ is 0.5, the equilibrium state has not been reached yet. Further, the σ_{xy} vs Iteration also depicts a similar situation that blue and cyan line are merge to 0 while red and green line are still apart from 0 after 8000 iterations which indicates the equilibrium state may have not been reached yet.



(a) $\lambda = 8$, Final configuration

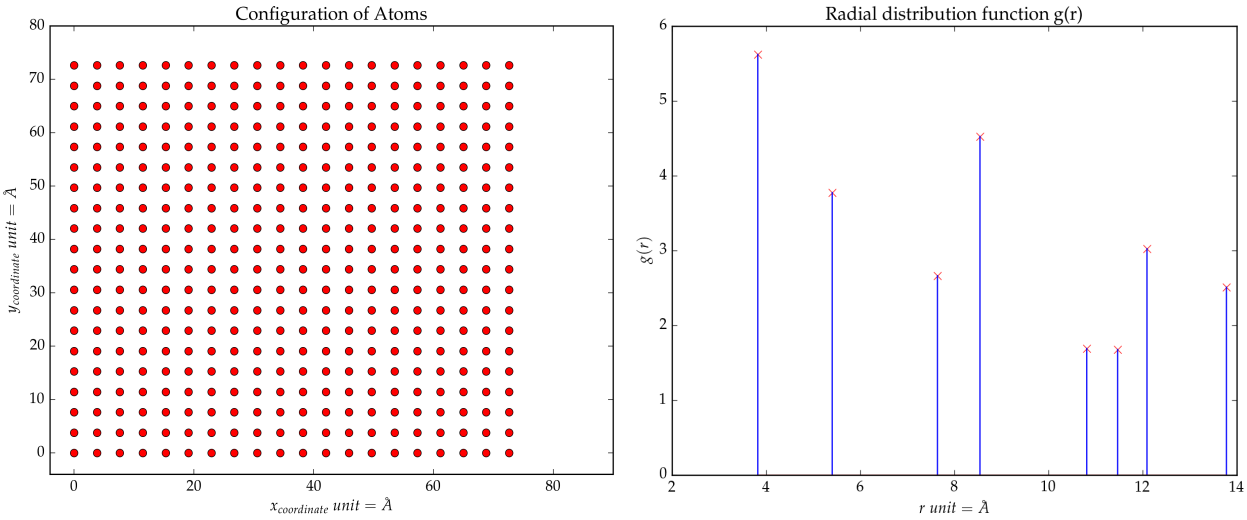


(b) $\lambda = 8$, Energy vs N



(c) $\lambda = 8$, Force vs N

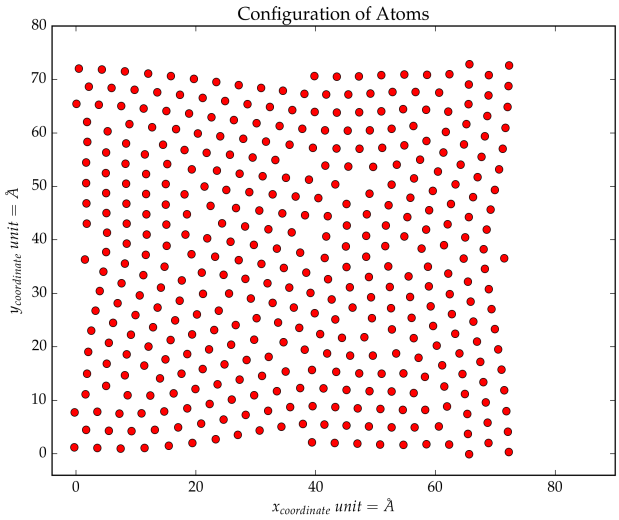
Figure 1: $\lambda = 8$, the too big λ case



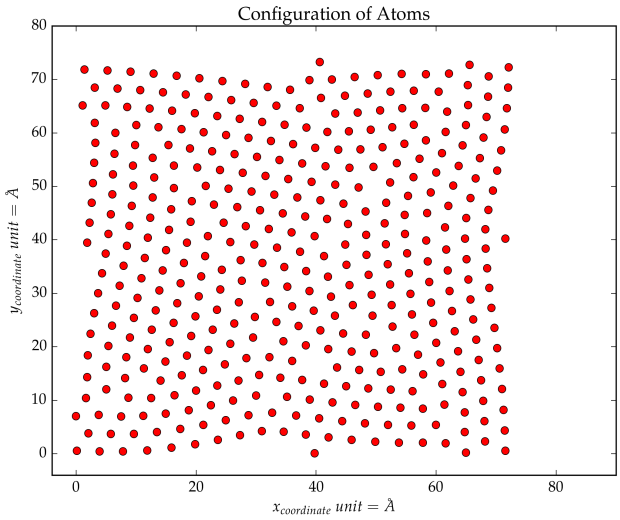
(a) Initial Configuration

(b) Initial RDF

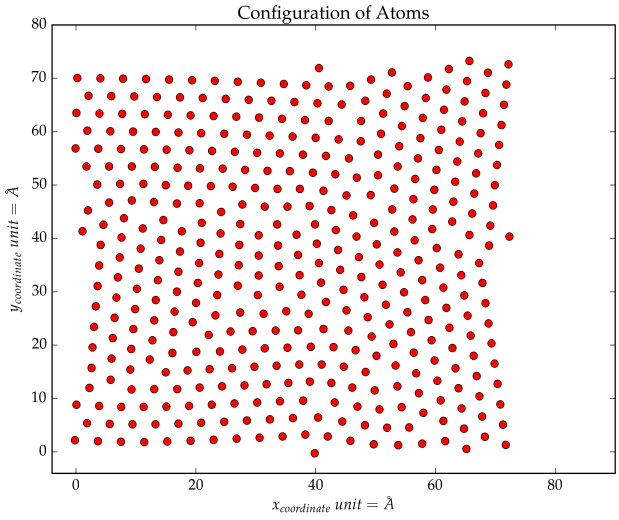
Figure 2: Initial state



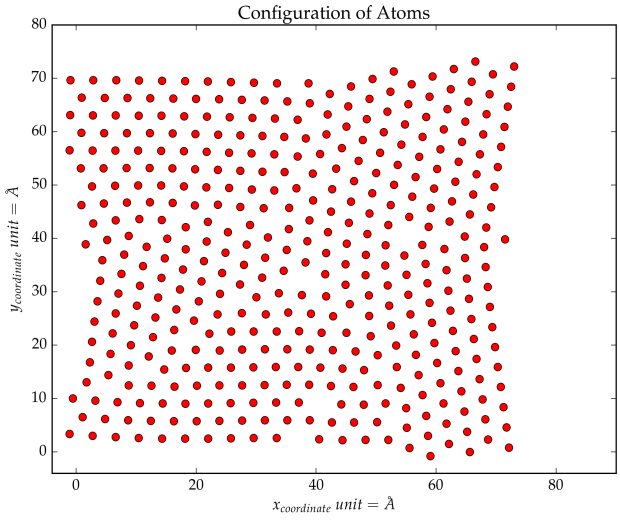
(a) $\lambda = 0.5$, Final configuration



(b) $\lambda = 1$, Final configuration

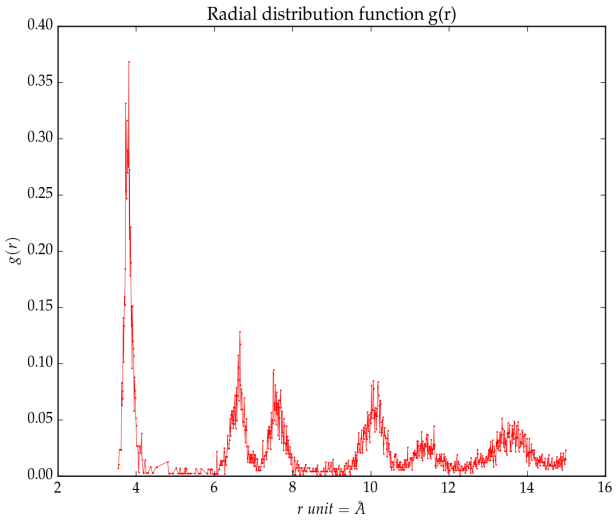


(c) $\lambda = 1.5$, Final configuration

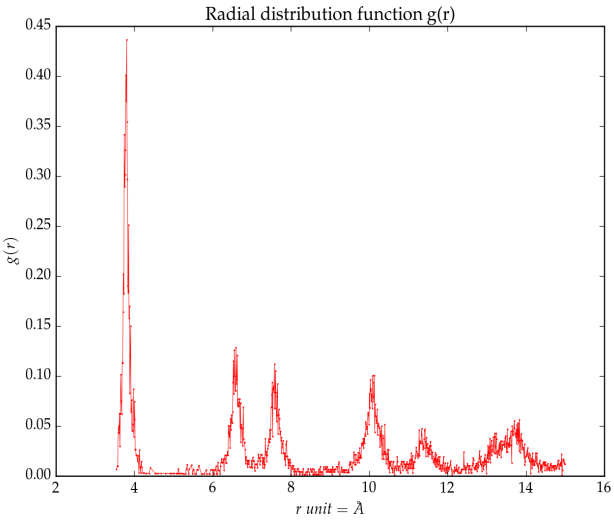


(d) $\lambda = 2$, Final configuration

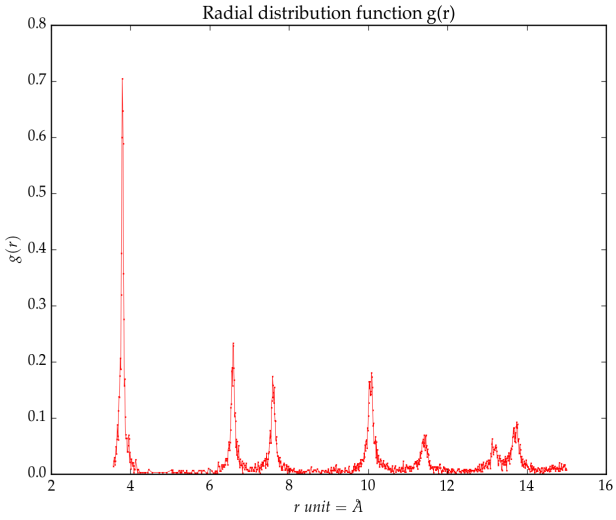
Figure 3: Final Configurations



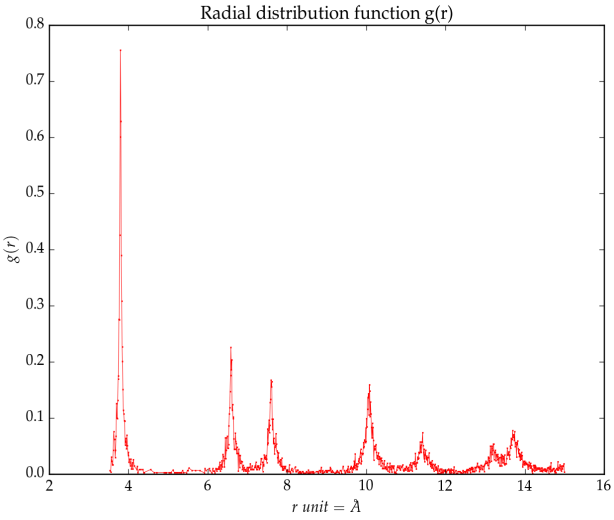
(a) $\lambda = 0.5$, Final RDF



(b) $\lambda = 1$, Final RDF



(c) $\lambda = 1.5$, Final RDF



(d) $\lambda = 2$, Final RDF

Figure 4: Final RDF

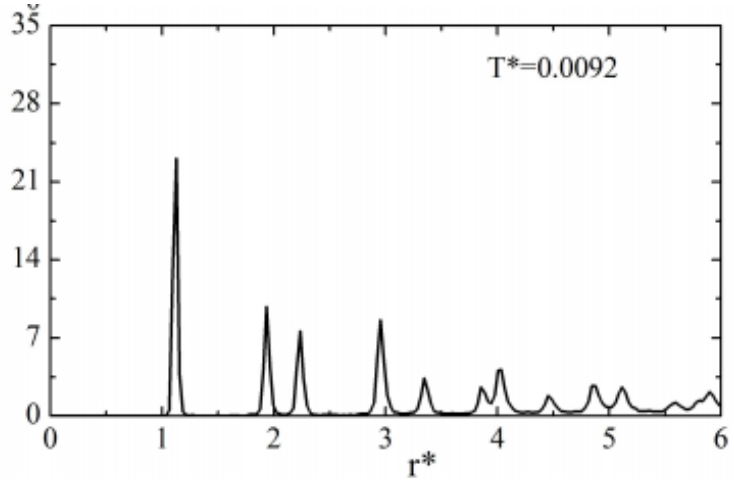
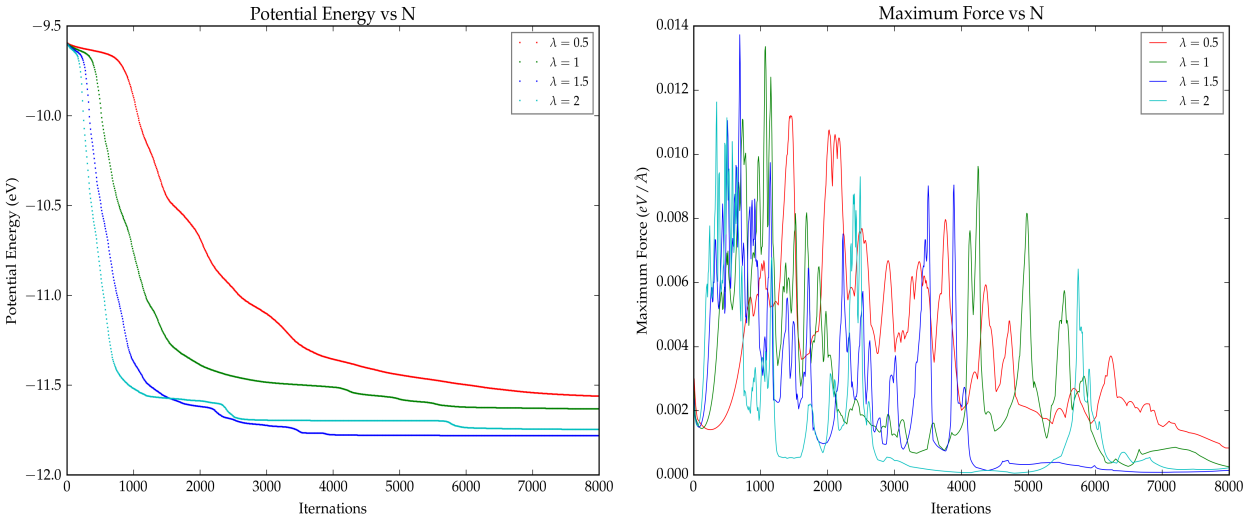


FIG. 1. Radial Distribution Function, IE parameters and ρ^*= 0.604 which corresponds to crystalline (diamond) silicon density.

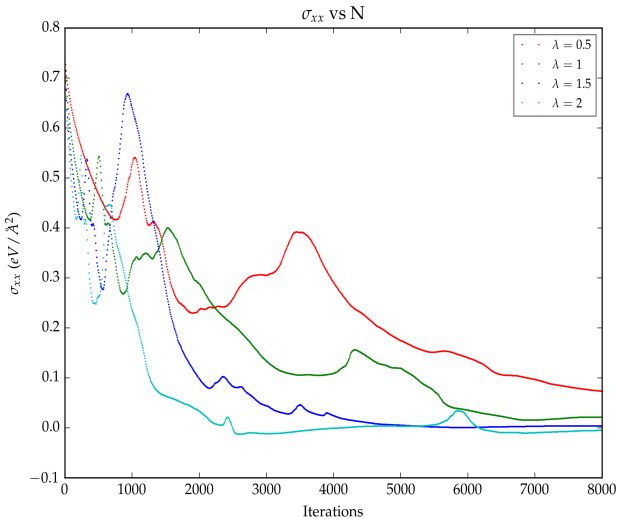
Figure 5: Reference RDF for diamond



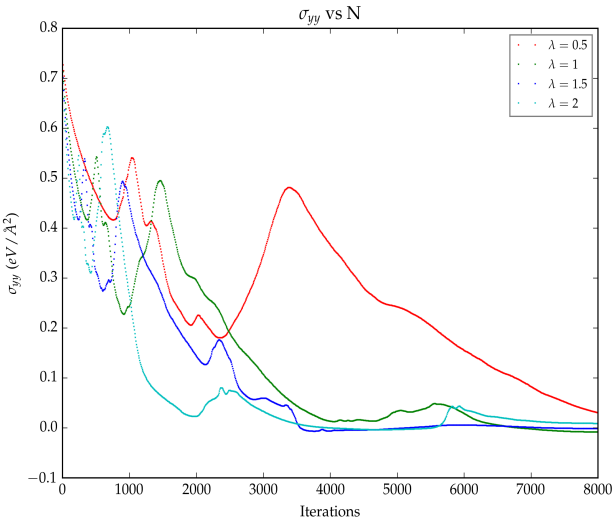
(a) Potential Energy vs Iteration

(b) Maximum Force vs Iteration

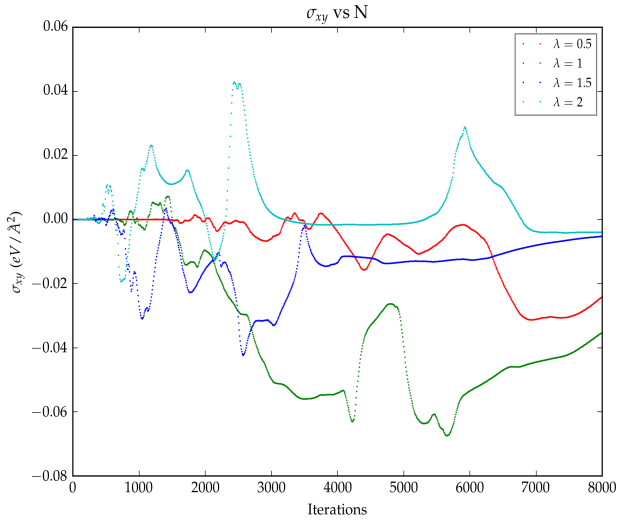
Figure 6: F,E vs N



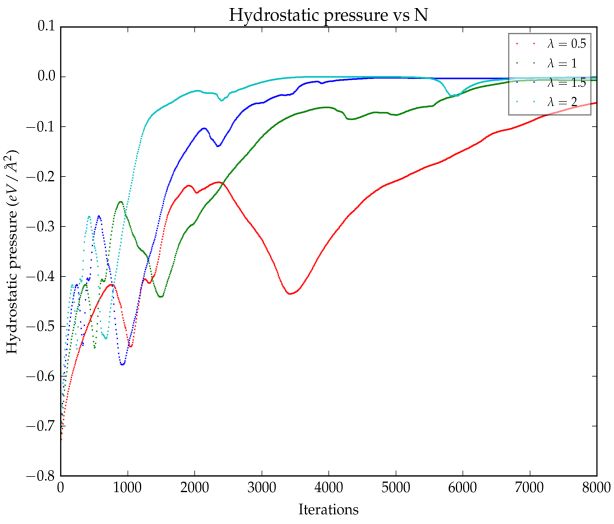
(a) σ_{xx} vs Iteration



(b) σ_{yy} vs Iteration



(c) σ_{xy} vs Iteration



(d) Hydrostatic Pressure vs Iteration

Figure 7: Variable vs N

2.2 Evolution of lambda = 1.5

In this section, we will take the $\lambda = 1.5$ as an example to illustrate the evolution of this relaxation simulation driven by our SD algorithm. We evaluate the evolution by taking a look at the block configuration, heatmap for atomic stress, as well as the RDF of the configuration on 0th, 250th, 500th, 1000th, 2000th and 8000th Iterations.

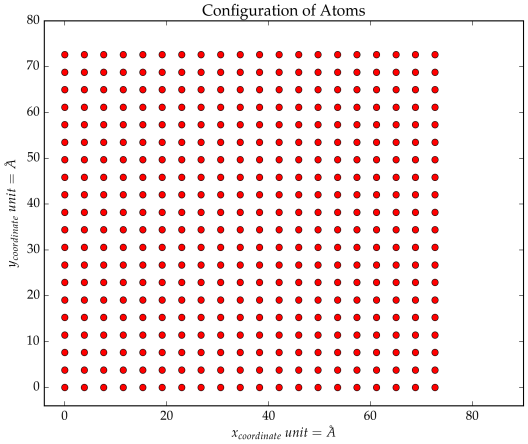
First of all, **Figure 8 describes the evolution of the configuration of the block**, and as we see from Figure 8 (b), compared with the initial block, the configuration does not change much up to 250th Iterations. However, when it goes to 500th iteration, the atoms start to pack together and as the iteration goes, eventually the block of atoms approach a close packed structure. Starting from 2000th iteration, the structure of the block stop to vary too much (with only tiny little modifications) and stay with such configuration until the end of simulation. Compare Figure 8 (e) and (f), we see the two configurations are very much similar and this behavior is expected as we could see from the previous plot on Figure 6 (a): the potential energy drop dramatically from 0th to 2000th iteration while only change very little from 2000th to 8000th iteration. This indicates the equilibrium (minimum potential energy) is almost reached when it goes to 2000th iteration.

Figure 9 illustrates the RDF corresponding to each Iterations and it is clear that it agrees with the behavior on configuration plot, that before 250th Iteration, the RDF does not change much until it goes to 500th Iteration. After then, the RDF became blur and fuzzy which indicates the structure are changing. When it goes to 1000th Iteration, the structure of RDF start to stabilize with 6 peaks while is still kind of fuzzy. At 2000th Iteration, a 7 peak RDF structure appears and become sharper and clear. Finally at 8000th iteration, a clear 7 peak RDF has been reached and this RDF matches with the reference RDF plot for diamond from Figure 5, hence predicts we have reached a close packed structure.

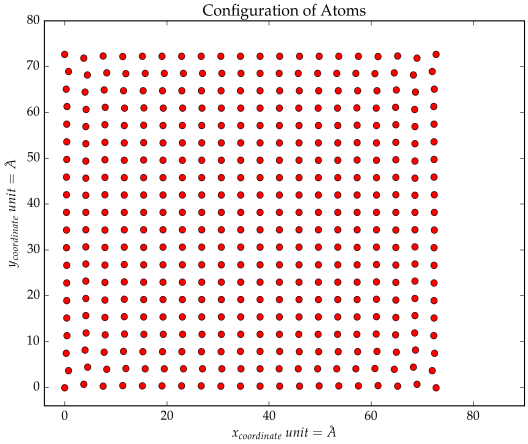
Lastly, **Figure 10, Figure 11, Figure 12 and Figure 13 are heat maps generated by the atomic stress and the hydrostatic pressure on each Iterations** and from where, we could visualize how this relaxation procedure process. We see all three components of the stress tensor σ_{xx}, σ_{yy}, σ_{xy} as well as the hydrostatic pressure P are trying to reach 0 (to be green) at the end of the simulation. Both the σ_{xx} and σ_{yy} heatmaps are starting from a red dominated (high stress) picture and end up with a green dominated picture (0 stress), where the majority of atoms have 0 stress and only a few atoms at the 'interface' (looks like grain boundary) experience stresses. This means the majority of the atoms have become more and more stable/relax. Similarly, the P heatmap starts from a blue dominated picture (negative pressure since it's a negative average of σ_{xx} and σ_{yy}) and as expected, end up with a green dominated picture. The σ_{xy} heatmap starts from a green dominated picture (0 stress), and as the simulation process, the variation of structure leads the color/value of σ_{xy} goes back (red) and force(blue), but eventually, when the equilibrium state established, the color/stress goes back to a a green dominated picture (0 stress). Finally the final green dominated picture of all 4 heatmaps reflects a stable and relaxed final configuration.

With Figure 8, Figure 9, Figure 10, Figure 11, Figure 12 and Figure 13 we see how our

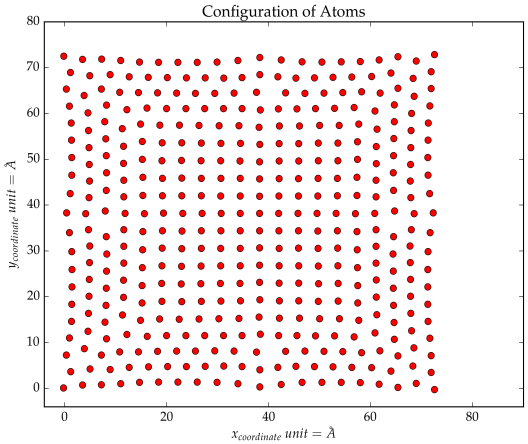
SD simulation changes the block and how the block evolves from a initially sparsely distributed structure into a finally close packed structure.



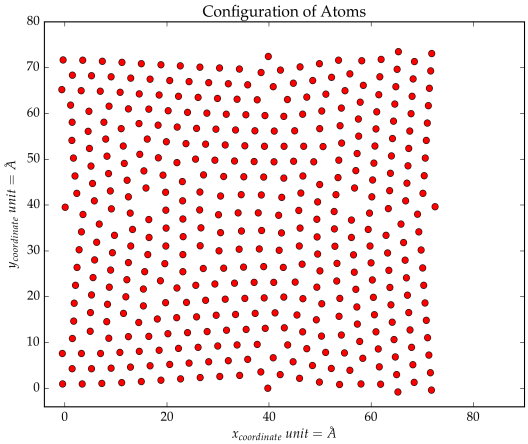
(a) Initial Configuration



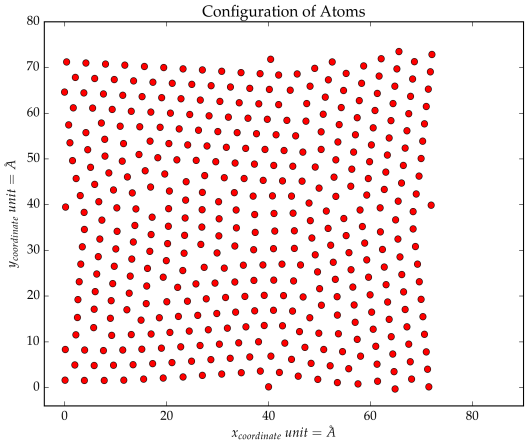
(b) Configuration at 250th iteration



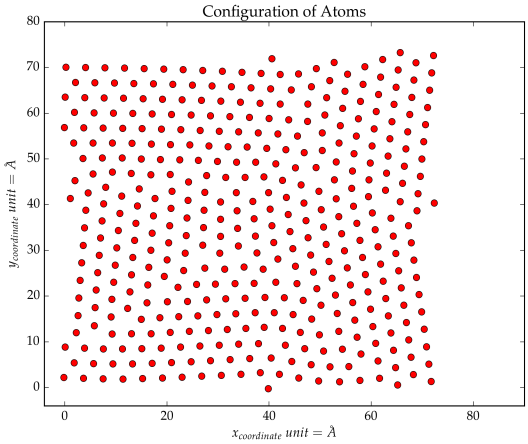
(c) Configuration at 500th iteration



(d) Configuration at 1000th iteration

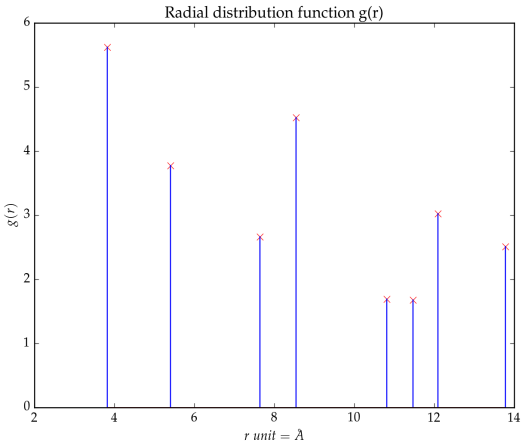


(e) Configuration at 2000th iteration

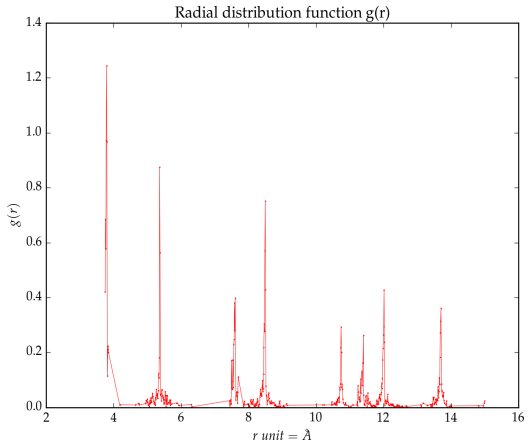


(f) Configuration at 8000th iteration

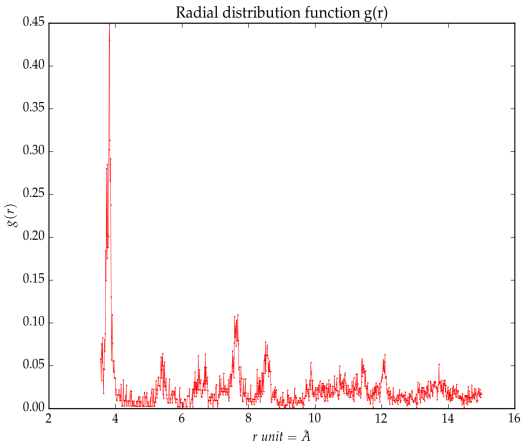
Figure 8: Evolution of configuration: $\lambda = 1.5$



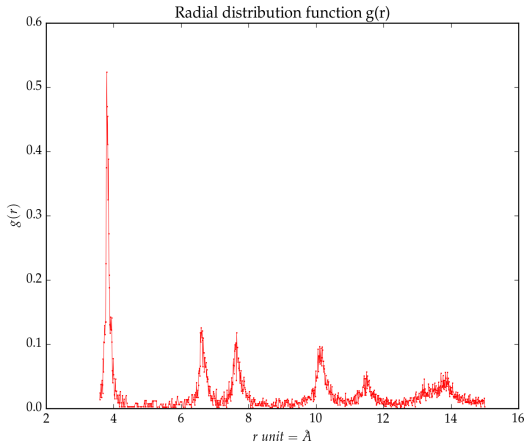
(a) Initial RDF



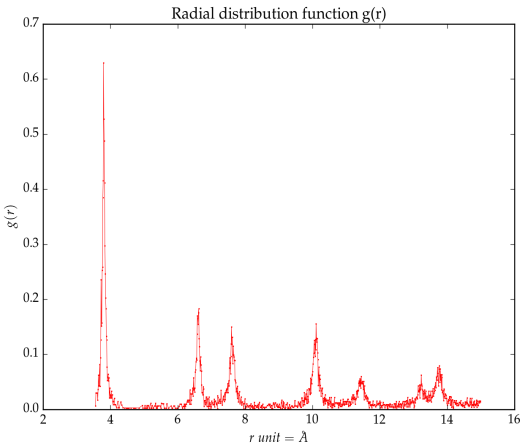
(b) RDF at 250th iteration



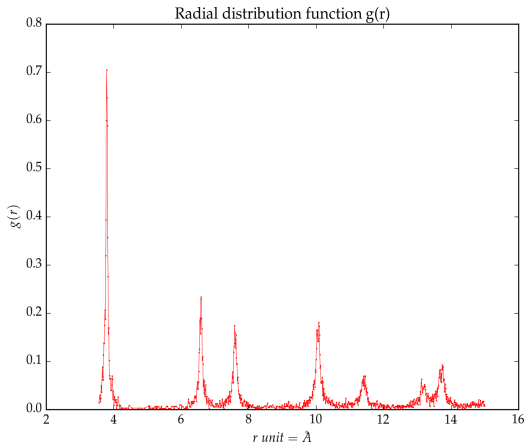
(c) RDF at 500th iteration



(d) RDF at 1000th iteration

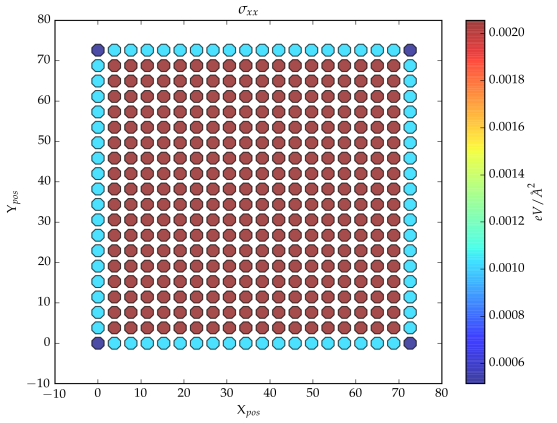


(e) RDF at 2000th iteration

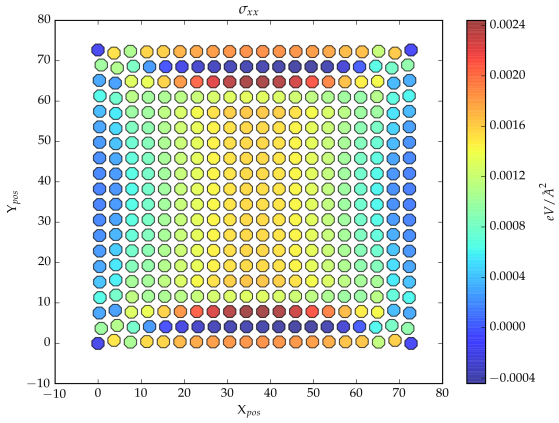


(f) Final RDF

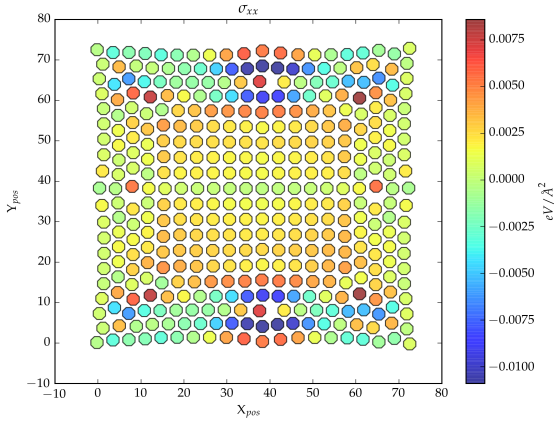
Figure 9: Evolution of RDF: $\lambda = 1.5$



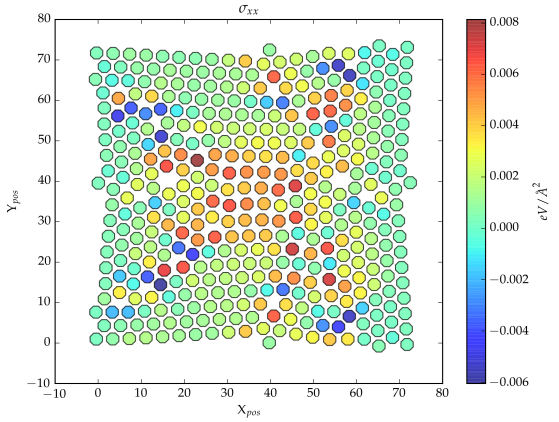
(a) Initial σ_{xx}



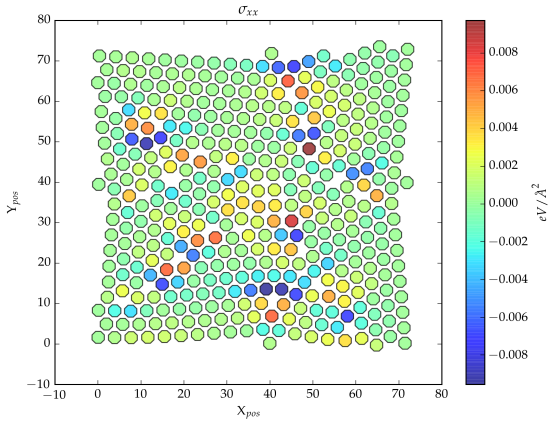
(b) σ_{xx} at 250th iteration



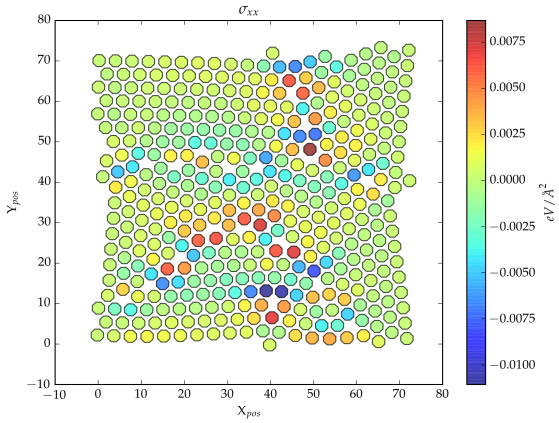
(c) σ_{xx} at 500th iteration



(d) σ_{xx} at 1000th iteration

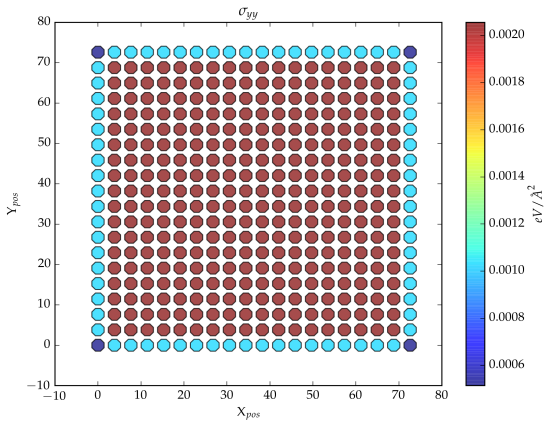


(e) σ_{xx} at 2000th iteration

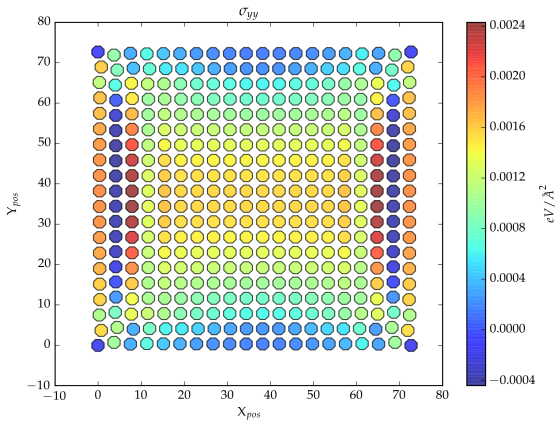


(f) σ_{xx} at 8000th iteration

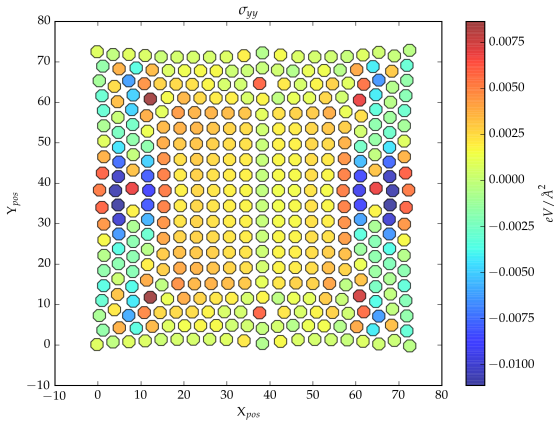
Figure 10: Evolution of σ_{xx}: $\lambda = 1.5$



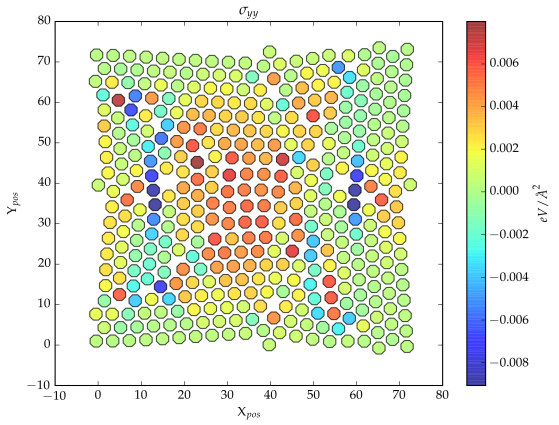
(a) Initial σ_{yy}



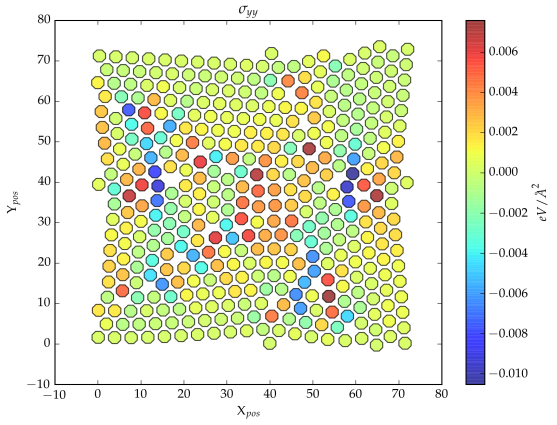
(b) σ_{yy} at 250th iteration



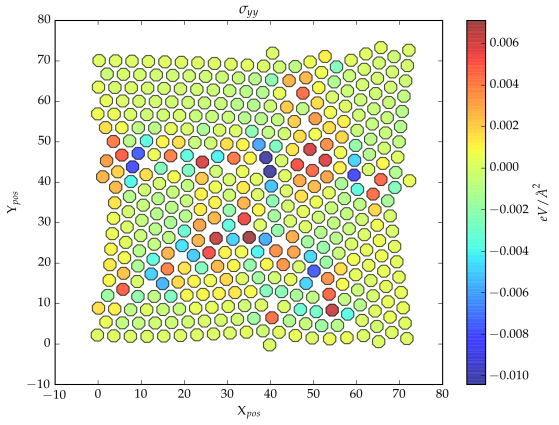
(c) σ_{yy} at 500th iteration



(d) σ_{yy} at 1000th iteration

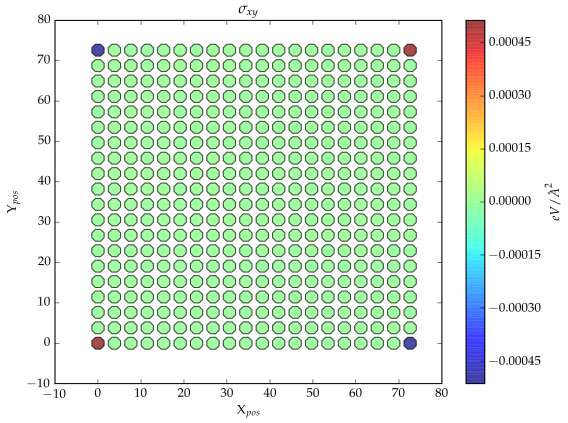


(e) σ_{yy} at 2000th iteration

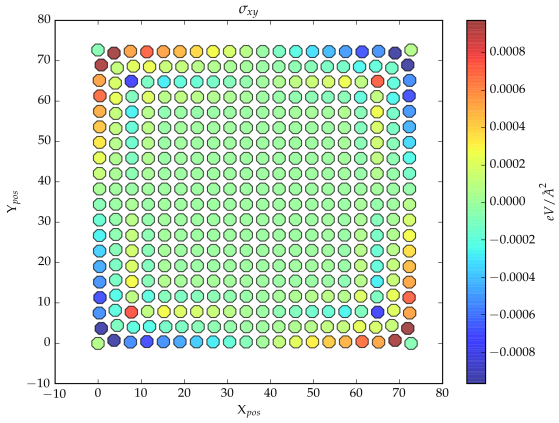


(f) σ_{yy} at 8000th iteration

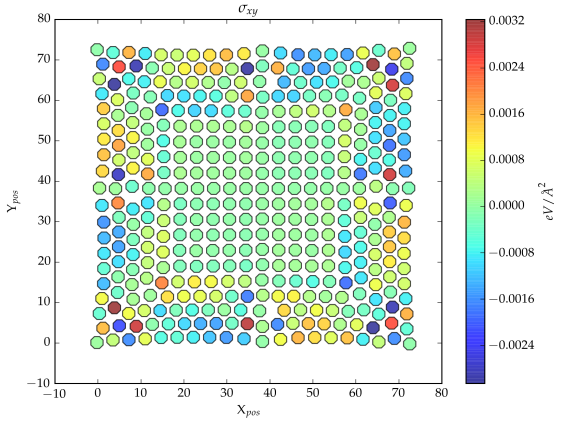
Figure 11: Evolution of σ_{yy}: $\lambda = 1.5$



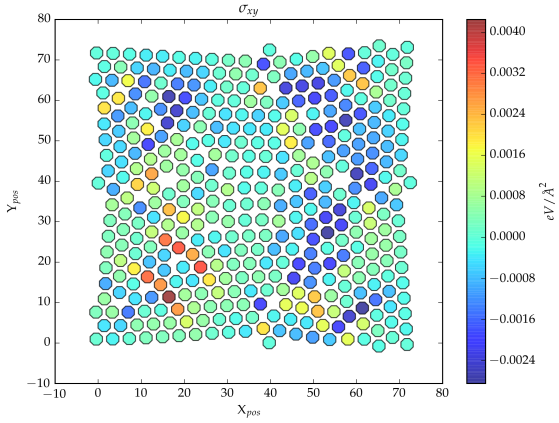
(a) Initial σ_{xy}



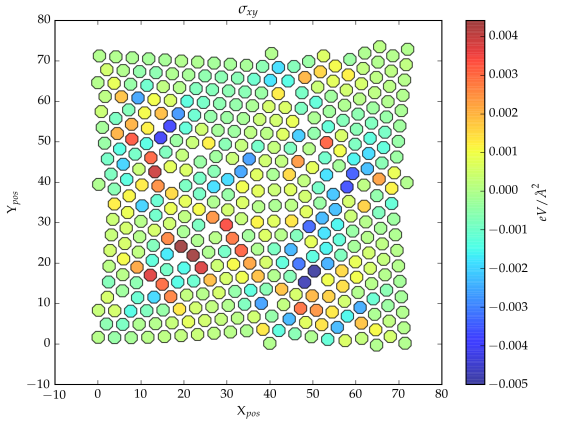
(b) σ_{xy} at 250th iteration



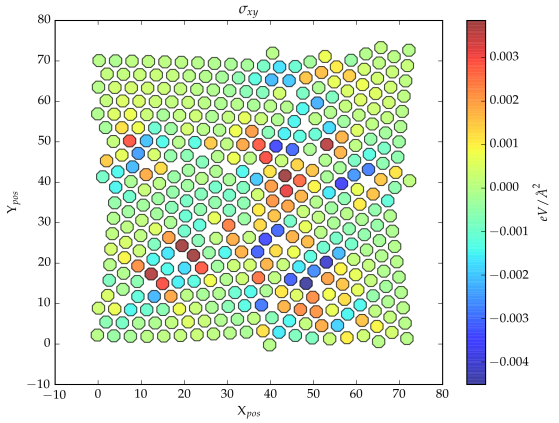
(c) σ_{xy} at 500th iteration



(d) σ_{xy} at 1000th iteration

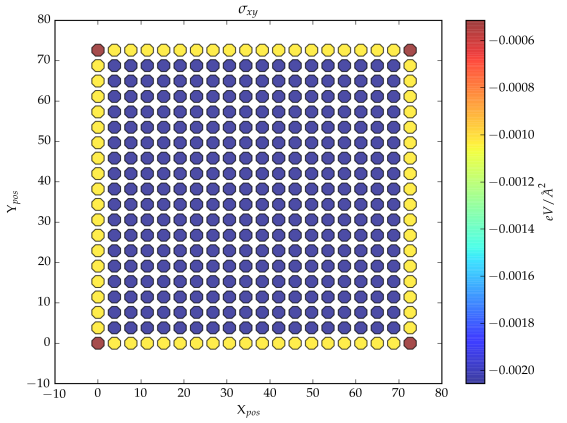


(e) σ_{xy} at 2000th iteration

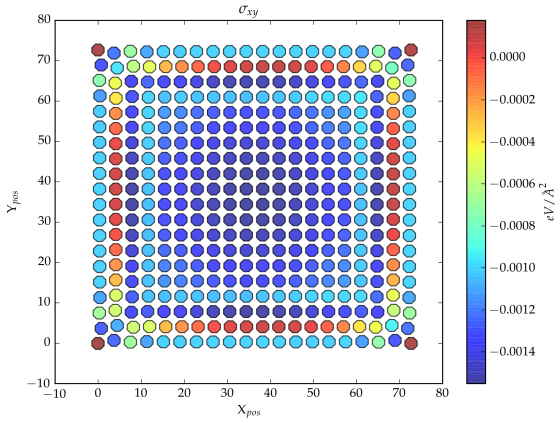


(f) σ_{xy} at 8000th iteration

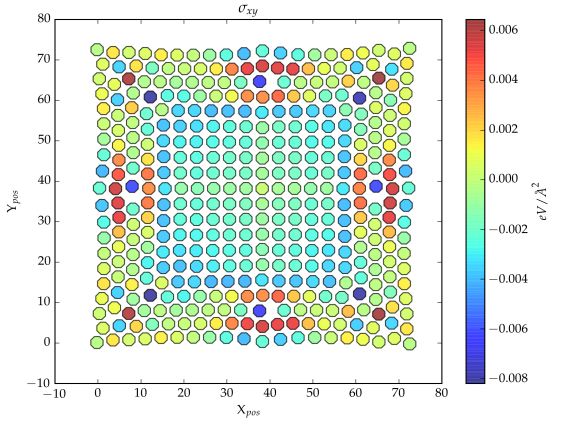
Figure 12: Evolution of σ_{xy}: $\lambda = 1.5$



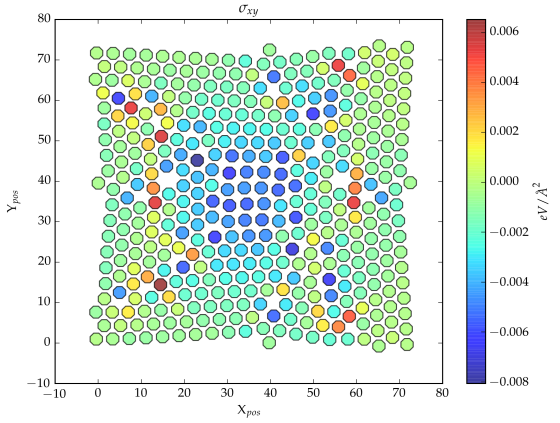
(a) Initial P



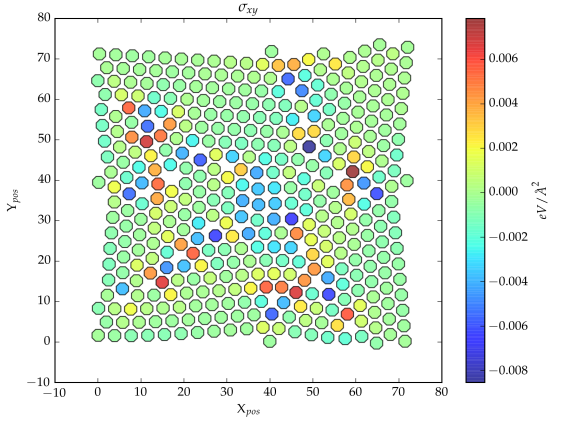
(b) P at 250th iteration



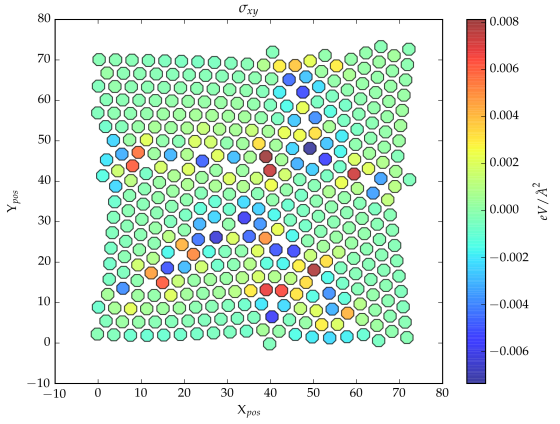
(c) P at 500th iteration



(d) P at 1000th iteration



(e) P at 2000th iteration



(f) P at 8000th iteration

Figure 13: Evolution of hydrostatic pressure P: $\lambda = 1.5$

3 Conclusion

1. **The choice of λ can't be too big or too small**. When it is too big, λ=8 in this work, the atom get stuck and failed to reach the relaxation state. When it is too small, λ=0.5 in this work, it takes too long to reach the equilibrium state, and in our case, even 8000 Iteration is not good enough for the block to relax. (all the hydrostatic pressure, stress and maximum force on atom for λ=0.5 have not reached 0 yet)

2. Compare the results from λ = 0.5, 1, 2 and 1.5 (which I have tried many sets of simulations and they all yields the same result), that the one with λ = 1.5 reaches the lowest final potential (-11.7813 eV) energy and the one with λ = 2 gets the 2nd lowest potential energy. For both λ=2 and 1.5 simulations, the hydrostatic pressure, stresses and maximum force on atoms have all approaching to 0, hence indicates the final potential is the minimum potential for each simulation. Since the final minimum potential energy from λ = 1.5 and 2 differs from each other, it shows that a different λ may result a slightly different final potential energy hence **the minimum potential energy we have reached is actually the local minimum and is depending on the choice of λ. Due to this, an even lower minimum potential energy might be reached with a proper choice of λ.** (we have only tested 4 λ values in this work).

3. Many factors indicate **the final potential energy we have reached is the local minimum potential energy**, for example, **the maximum Force on a single atom, the total hydrostatic pressure and the total stresses are all approaching to 0**. Since we have Lenard Jones interaction only and assumed there is no kinetics involved, there is no source to increase the potential energy anymore such that the potential we have reached is the local minimum.

4. The RDF become more and more sharp and clear as the block reaches to relaxation/its equilibrium/lowest potential energy and our final RDF matches with the reference RDF for diamond. **This indicates a closed packed structure have been formed at the final configuration**.

4 Appendix: source code

Firstly, thanks very much for reading the source code. (I know it is always a pain to do so...) 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
// HW3 Molecular statics relaxation
// 2-D Block
// Author: Yuding Ai
// Date: 2017.03.01

#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
 *
 * Since I misunderstood assignment 2 and kind of get lost last
 * time, (I thought the block is mapping into a lattice space so
 * every atom sits on a lattice site, like a pixel, now I realize
 * it's completely wrong.) I have revised the whole structure of
 * my 2dblock. (hopefully it's getting better this time)
 *
 * Here is a brief description of it:
 *
 * A. To compile the file: g++ -std=gnu++11 main.cpp 2dblock.cpp -o run
 * B. To run the compiled file: ./run
 * C. To get the plots: python hw3.py
 *
 * 0.Part of the implementation (the optimized algorithm for
 * get_neighbor_list for example) is inspired by Our TA Spencer's
 * nice reference code:assignment2Demo.f90;
 *
 * 1. Same as before, I treat this whole 2dblock as a single object
 * with certain amount of atoms in it. In this assignment, with 400 atoms.
 * The class 2dblock have 5 attributes:
 * w: width of the block; (#of atoms each raw at initial config);
 * h: height ot the block;(#of atoms each col at initial config);
 * n: number of atoms.
 * neighbor_list: a 'list' of neighborlist contains neighbors for each atom.
 * atomlist: a list of atoms, contains all the infomations we need so far
 * for each atom.
 *
 * 2. Each atom is a '7D vector' (an array with 7 components) associated
 * with a index and stored into an atomlist; such that Atom(xposition,
 * yposition, sigmaxx, sigmayy, sigmaxy, F_x, F_y).
 * so: std::array<double,7> atom;
 * [Remark: I might want to make class for my atom for latter
 * assignment if needed to, but for now, I will keep it as simple as
 * a '7D vector']
 *
 * 3.Methods (subroutine):
 *
 * 3.1 method get_neighbor_list(r), where r= neighCut except for
 * calculate rdf, in rdf we set r = rdfMax. Update attribute
 * neighbor_list into current configuration.
 *
 * 3.2 method get_atomic_stress(), update the stress for each atom
```

```
 * and corresponding to current configuration and load it to the
 * class arribute: atomlist.
 *
 * 3.3 method get_distance(idx1,idx2) return the distance of 2 atom
 *     method get_xdistance(idx1,idx2) return the xdistance of 2 atom
 *     method get_ydistance(idx1,idx2) return the ydistance of 2 atom
 *
 * 3.4 method rdf(filename),calculate the rdf and store it into a
 * txt file named filename.
 *
 * 3.5 method out_config(filename), output the current configuration
 * including xpos,ypos,sigma_xx,sigma_yy,sigma_xy for each atom
 * for into a txt file named filename for latter plotting
 *
 * 3.6 method total_energy(), fist update the neighbor_list and then
 * calculate the total energy under the corresponding configuration.
 *
 * 3.7 method calc_force(), calculate the force for all atoms and assign
 *     it to each atom.
 *     method calc_single_force(idx), calculate the force for a single
 *     atom with index idx
 *     method qcalc_mforce(), quickly calculate the maximum force of the
 *     corresponding configuration.
 *
 * 3.8 method calc_total_stress_pressure(), calculate the net stresses
 * of the corresponding configuration.
 *
 * 3.9 method psi(r) compute Lenard Jones potential between 2 atoms
 *     that are separated by r.
 *     method dpsi(r) compute derivative of Lenard Jones potential
 *     between 2 atoms that are separated by r.
 *
 * 4.Core molecular simulation method SD(steepest descent):
 * SD(lambda, iteration, filename) perform a simulation with SD method and
 * output the simulation data into a txtfile named filename
 *
 * All in all, thanks very much for reading this script and a big
 * special thanks to Spencer's reference code which is really helpful
 * and answers a lot of questions I had before.
 *
 * Again, 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;


class block
{
    private:
        int w;           // Width;
        int h;           // Height;
        int n;           // number of atoms;

        // each atom is represented by an array
        array<array<double,7>,400> atomlist;
        // Once the idx is known, all the info about idxth atom is known
        // because we could reach such atom by: block[idx]

        //a list of neighbor lists of each atom
        array<vector<int>,400> neighbor_list;

    public:
        ///------------constructor---------------------------------------
        //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
        ///-------------------------------------------------------------
        block(int x,int y,int N);


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

        int getN();

        // 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]
        // for example: its position is (block[idx][0],block[idx][1]) and its
        // stresses are (block[idx][2],block[idx][3],block[idx][4])
        array<double,7> &operator[](int index);


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

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

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

        //------------get_neighbor_list---------------------------------
```

```
        // take a single argument r, which is neighCut in this assignment
171     // and will update neighbor_list of current configuration.
        // Using Spencer's trick to reduce the computational complicity
173     // ------------------------------------------------------------
        void get_neighbor_list(double r);
175
        //------------get_atomic_stress--------------------------------
177     //take no argument and compute the atomic stresses of all the
        //atoms and assign it to each atom.
179     // ------------------------------------------------------------
        void get_atomic_stress();
181
        //------------rdf() -------------------------------------------
183     //Get the radial distribution and output into a txt file
        // ------------------------------------------------------------
185     void rdf(string filename);

187

        //------------out_config()-------------------------------------
189     // output the configuration of atoms for latter plotting
        // and visualization
191     // ------------------------------------------------------------
        void out_config(string filename);
193
        //------------total_energy()-----------------------------------
195     // compute the total energy of a configuration by sum over
        // the potential energy of all atoms
197     // E_tot = 1/2 sum_{i,j} psi(r_{i,j})
        // ------------------------------------------------------------
199     double total_energy();

201     //------------calc_force() & calc_single_force()----------------
        //compute the force of a configuration for each atom by:
203     //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
205     //then find and return the maximum force;
        //
207     //calc_single_force()
        //compute the net force on a single atom and assign such
209     //force onto atomlist[idx][5] and atomlist[idx][6]
        //
211     //qcalc_mforce() calculate the maximum force a current atomlist
        //by simply sort the force component
213     // ------------------------------------------------------------
        double calc_force();
215     void calc_single_force(int idx);
        double qcalc_mforce();
217

219     //------------calc_total_stress_pressure()---------------------
        //compute the total stress sigma_xx, simga_yy and simga_xy and
221     //the hydrostatic pressure P. Store the 4 values into an array
        //such that (sum_simga_xx,sum_simga_yy,sum_simga_xy,P)
223     //------------------------------------------------------------
```

```
        array<double,4> calc_total_stress_pressure();

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

};

/// 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
// HW3 Molecular statics relaxation
// 2-D Block
// Author: Yuding Ai
// Date: 2017.03.01

#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,7> atom;
        atom[0] = xpos;
        atom[1] = ypos;
        atom[2] = atom[3]=atom[4] = atom[5] = atom[6]=0.0;
        atomlist[i]=atom;
    }

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

    // calculate the initial atomic stresses
    get_atomic_stress();
```

```
35      // calculate the initial Force on each atom
        calc_force();
37  }

39  ///-------------------------------------------------------------
    /// getter (subroutines)
41  ///-------------------------------------------------------------

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

45  array<double,7> &block::operator[](int index){return atomlist[index];}

47
    double block::get_distance(int idx1, int idx2) const {
49      array <double,2> coor1 = {{atomlist[idx1][0],atomlist[idx1][1]}};
        array <double,2> coor2 = {{atomlist[idx2][0],atomlist[idx2][1]}};
51      double dis;
        dis =sqrt(pow(coor1[0]-coor2[0],2) + pow(coor1[1] - coor2[1],2));
53      return dis;
    }
55
    double block::get_xdistance(int idx1, int idx2) const {
57      array <double,2> coor1 = {{atomlist[idx1][0],atomlist[idx1][1]}};
        array <double,2> coor2 = {{atomlist[idx2][0],atomlist[idx2][1]}};
59      double dis;
        dis =coor1[0]-coor2[0];
61      return dis;
    }
63
    double block::get_ydistance(int idx1, int idx2) const {
65      array <double,2> coor1 = {{atomlist[idx1][0],atomlist[idx1][1]}};
        array <double,2> coor2 = {{atomlist[idx2][0],atomlist[idx2][1]}};
67      double dis;
        dis =coor1[1]-coor2[1];
69      return dis;
    }
71
    void block::get_neighbor_list(double r){
73      //first remove previous neighbor list:

75      for (int i = 0; i < n; i++) {
            neighbor_list[i].clear();
77      }
        //then load the new neighbor list
79      // Thanks to Spencer's nice reference code for assignment 2, here
        // we apply the same nice trick to get rid of double counting and
81      // cut computational complexity by half

83      for (int i = 0; i < n; i++) {
            for (int j = i+1; j < n; j++) {
85              // for this assignment, r = neighCut
                if(get_distance(i,j) <=r){
87                  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){
143         c++;
        }
145     else{
            //rlist is a list of possible distance of all neighbors
147         //sorted in increment order
            Rlist.push_back(d);
149         //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
151         c = 0;
            d = dislist[i];
153     }

155     if(i == dislist.size()-1){
            //rlist is a list of possible distance of all neighbors
157         //sorted in increment order
            Rlist.push_back(d);
159         //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
161         c = 0;
            d = dislist[i];
163     }
    }
165

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

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

    //last reset the neighbor list to r = neighCut
181 //since we modified the r to be rdfMax every time when
    //we calculate the rdf
183 get_neighbor_list(neighCut);
}
185

187 void block::out_config(string filename){

189     stringstream st;
    // record xpos, ypos and stresses of each atom into a txt file
191 // file for latter plotting
    for (int i = 0; i < n; i++) {
193     st << atomlist[i][0]<<" "<<atomlist[i][1]<<" "<<atomlist[i][2]<<" ";
```

```
        st << atomlist[i][3]<<" "<<atomlist[i][4]<<endl;
    }

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

}
double block::total_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;
}

void block::SD(double lambda,int it,string filename){
    stringstream st;
    int j = 0;
    int k = 0;
    while(j<it){
        for (int i = 0; i < n; i++) {
            // get the force on each atom
            calc_single_force(i);
            // Move each atom following the direction
```

```
            // of it's force
303         atomlist[i][0] += lambda*atomlist[i][5];
            atomlist[i][1] += lambda*atomlist[i][6];
305     }

307     if(it>=1000){
            if(k==it/1000){
309             //first update the neighborlist
                get_neighbor_list(neighCut);
311             //update the stresses
                get_atomic_stress();
313             //calc hydrostatic pressure
                array<double,4> sp = calc_total_stress_pressure();
315             //calc total energy
                double E = total_energy();
317             //calc maximum force
                double F = qcalc_mforce();
319             //record the state/configuration data
                st<<j<<"   "<<E<<"   "<<sp[0]<<"  "<<sp[1]<<"  ";
321             st<<sp[2]<<"  "<<sp[3]<<"  "<<F<<endl;
                k = 0;
323         }
        }
325     else{
            //first update the neighborlist
327         get_neighbor_list(neighCut);
            //update the stresses
329         get_atomic_stress();
            //calc hydrostatic pressure
331         array<double,4> sp = calc_total_stress_pressure();
            //calc total energy
333         double E = total_energy();
            //calc maximum force
335         double F = qcalc_mforce();
            //record the state/configuration data
337         st<<j<<"   "<<E<<"   "<<sp[0]<<"  "<<sp[1]<<"  ";
            st<<sp[2]<<"  "<<sp[3]<<"  "<<F<<endl;
339     }

341     j++;
        k++;
343 }
    ofstream myfile(filename);
345 string data = st.str();
    myfile<< data;
347 myfile.close();

349 }
    ///------------------------------Outside block class ------------------
351 double psi(double r){
        double result;
353     //  r >= rCut
        if(r>=rCut){result = 0;}
355     //  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
// HW3 Molecular statics relaxation
// Main function
// Author: Yuding Ai
// Penn ID: 31295008
// Data: 2017.03.01

#include "2dblock.h"


int main() {

    block bloc1(20,20,400);
    // output the original configuration and rdf
    bloc1.rdf("rdf.txt");
    bloc1.out_config("config.txt");

    // Now examine on lambda = 0.5,1,1.5 and 2 seperately

    // ---------------lambda = 0.5 -------------
    bloc1.SD(0.5,8000,"l=05_vsN.txt");
    bloc1.out_config("l=05_config.txt");
    bloc1.rdf("l=05_rdf.txt");

    // ---------------lambda = 1 ---------------
    block bloc2(20,20,400);
    bloc2.SD(1,8000,"l=1_vsN.txt");
    bloc2.out_config("l=1_config.txt");
    bloc2.rdf("l=1_rdf.txt");

    // ---------------lambda = 1.5 -------------
    block bloc3(20,20,400);
    bloc3.SD(1.5,8000,"l=15_vsN.txt");
    bloc3.out_config("l=15_config.txt");
    bloc3.rdf("l=15_rdf.txt");

    // ---------------lambda = 2 ---------------
```

```
    block bloc4(20,20,400);
    bloc4.SD(2,8000,"l=2_vsN.txt");
    bloc4.out_config("l=2_config.txt");
    bloc4.rdf("l=2_rdf.txt");

    // ---------------lambda = 8 ---------------
    block bloc5(20,20,400);
    bloc5.SD(8,8000,"l=8_vsN.txt");
    bloc5.out_config("l=8_config.txt");
    bloc5.rdf("l=8_rdf.txt");

    // --------study the evolution -------------
    // ---------------lambda = 1.5 -------------
    block bloc6(20,20,400);

    bloc6.SD(1.5,250,"250vsN.txt");
    bloc6.out_config("250config.txt");
    bloc6.rdf("250rdf.txt");

    bloc6.SD(1.5,250,"500vsN.txt");
    bloc6.out_config("500config.txt");
    bloc6.rdf("500rdf.txt");

    bloc6.SD(1.5,500,"1000vsN.txt");
    bloc6.out_config("1000config.txt");
    bloc6.rdf("1000rdf.txt");

    bloc6.SD(1.5,1000,"2000vsN.txt");
    bloc6.out_config("2000config.txt");
    bloc6.rdf("2000rdf.txt");

    return 0;
}
```

Listing 3: main.cpp

```
# hw3.py
# HW3 Molecular statics relaxation
# Author: Yuding Ai
# Date: 2017.03.01

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"$eV/{\AA}^2$")
    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"$eV/{\AA}^2$")
    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"$eV/{\AA}^2$")
    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"$eV/{\AA}^2$")
    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"$\sigma_{xy}$"
    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 plot_vsN(tx1,tx2,tx3,tx4,tx5,EN,SxxN,SyyN,SxyN,PN,FN,E8,F8):
    '''make plots for stress vs N; potential energy vs N
    maximum Force vs N and hydrostatic pressure vs N'''

    #-----------------------------------------------------------------
    # step 1 load files
    #-----------------------------------------------------------------
    #--------------------lambda = 0.5 ----------------------------
    N = [] # a list of interation
    E = [] # a list of PE
    SXX = [] #sigma_xx
    SYY = [] #sigma_yy
    SXY = [] #sigma_xy
    P = [] #hydrostatic pressure
    F = [] #force
    with open(tx1,"r") as file:
        for line in file:
            words = line.split()
            n = float(words[0]) #take the value
            e = 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 = float(words[5]) #take the value
            f = float(words[6]) #take the value

            N.append(n); #append x value into X
            E.append(e); #append y value into Y
            SXX.append(sxx); #append sxx value into SXX
            SYY.append(syy); #append syy value into SYY
            SXY.append(sxy); #append sxy value into SXY
            P.append(p);
            F.append(f);
```

```
180
    #--------------------lambda = 1 -----------------------------
182 N1= [] # a list of interation
    E1= [] # a list of PE
184 SXX1= [] #sigma_xx
    SYY1= [] #sigma_yy
186 SXY1= [] #sigma_xy
    P1= [] #hydrostatic pressure
188 F1= [] #force
    with open(tx2,"r") as file:
190     for line in file:
            words = line.split()
192         n = float(words[0]) #take the value
            e = float(words[1]) #take the value
194         sxx = float(words[2]) #take the value
            syy = float(words[3]) #take the value
196         sxy = float(words[4]) #take the value
            p = float(words[5]) #take the value
198         f = float(words[6]) #take the value

200         N1.append(n); #append x value into X
            E1.append(e); #append y value into Y
202         SXX1.append(sxx); #append sxx value into SXX
            SYY1.append(syy); #append syy value into SYY
204         SXY1.append(sxy); #append sxy value into SXY
            P1.append(p);
206         F1.append(f);

208 #--------------------lambda = 1.5 ---------------------------
    N2= [] # a list of interation
210 E2= [] # a list of PE
    SXX2= [] #sigma_xx
212 SYY2= [] #sigma_yy
    SXY2= [] #sigma_xy
214 P2= [] #hydrostatic pressure
    F2= [] #force
216 with open(tx3,"r") as file:
        for line in file:
218         words = line.split()
            n = float(words[0]) #take the value
220         e = float(words[1]) #take the value
            sxx = float(words[2]) #take the value
222         syy = float(words[3]) #take the value
            sxy = float(words[4]) #take the value
224         p = float(words[5]) #take the value
            f = float(words[6]) #take the value
226
            N2.append(n); #append x value into X
228         E2.append(e); #append y value into Y
            SXX2.append(sxx); #append sxx value into SXX
230         SYY2.append(syy); #append syy value into SYY
            SXY2.append(sxy); #append sxy value into SXY
232         P2.append(p);
            F2.append(f);
```

```
234
    #-------------------lambda = 2 -----------------------------
236 N3= [] # a list of interation
    E3= [] # a list of PE
238 SXX3= [] #sigma_xx
    SYY3= [] #sigma_yy
240 SXY3= [] #sigma_xy
    P3= [] #hydrostatic pressure
242 F3= [] #force
    with open(tx4,"r") as file:
244     for line in file:
            words = line.split()
246         n = float(words[0]) #take the value
            e = float(words[1]) #take the value
248         sxx = float(words[2]) #take the value
            syy = float(words[3]) #take the value
250         sxy = float(words[4]) #take the value
            p = float(words[5]) #take the value
252         f = float(words[6]) #take the value

254         N3.append(n); #append x value into X
            E3.append(e); #append y value into Y
256         SXX3.append(sxx); #append sxx value into SXX
            SYY3.append(syy); #append syy value into SYY
258         SXY3.append(sxy); #append sxy value into SXY
            P3.append(p);
260         F3.append(f);

262 #-------------------lambda = 8 -----------------------------
    N4= [] # a list of interation
264 E4= [] # a list of PE
    F4= [] #force
266 with open(tx5,"r") as file:
        for line in file:
268         words = line.split()
            n = float(words[0]) #take the value
270         e = float(words[1]) #take the value
            p = float(words[5]) #take the value
272         f = float(words[6]) #take the value

274         N4.append(n); #append x value into X
            E4.append(e); #append y value into Y
276         F4.append(f);

278 #-------------------------------------------------------------
    #step 2
280 #-------------------------------------------------------------

282 #plot the plots

284 #-------------------plot Potential energy vs N---------------

286 fig = plt.figure()
```

```
288     plt.plot(N,E,'rx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 0.5$
        ')
        plt.plot(N,E1,'gx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 1$'
        )
290     plt.plot(N,E2,'bx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 1.5
        $')
        plt.plot(N,E3,'cx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 2$'
        )
292
        leg = plt.legend(prop={'size':10})
294     leg.get_frame().set_alpha(0.5)
        xlabel = r'Iternations'
296     ylabel = r'Potential Energy (eV)'
        plt.xlabel(xlabel)
298     plt.ylabel(ylabel)
        plt.title(r'Potential Energy vs N')
300     fig.savefig(EN,dpi = 300, bbox_inches ='tight')


302
        #--------------------plot stresses vs N ----------------------
304     # Sxx
        fig= plt.figure()
306     plt.plot(N,SXX,'rx',linewidth = 0.3,markersize = 1,label = r'$\lambda =
        0.5$')
        plt.plot(N,SXX1,'gx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 1
        $')
308     plt.plot(N,SXX2,'bx',linewidth = 0.3,markersize = 1,label = r'$\lambda =
        1.5$')
        plt.plot(N,SXX3,'cx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 2
        $')
310
        leg = plt.legend(prop={'size':10})
312     leg.get_frame().set_alpha(0.5)
        xlabel = r'Iterations'
314     ylabel = r'$\sigma_{xx}$ ($eV/\AA^{2}$)'
        plt.xlabel(xlabel)
316     plt.ylabel(ylabel)
        plt.title(r'$\sigma_{xx}$ vs N')
318     fig.savefig(SxxN,dpi = 300, bbox_inches ='tight')

320     # Syy
        fig= plt.figure()
322     plt.plot(N,SYY,'rx',linewidth = 0.3,markersize = 1,label = r'$\lambda =
        0.5$')
        plt.plot(N,SYY1,'gx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 1
        $')
324     plt.plot(N,SYY2,'bx',linewidth = 0.3,markersize = 1,label = r'$\lambda =
        1.5$')
        plt.plot(N,SYY3,'cx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 2
        $')
326
        leg = plt.legend(prop={'size':10})
328     leg.get_frame().set_alpha(0.5)
        xlabel = r'Iterations'
```

```
330     ylabel = r'$\sigma_{yy}$ ($eV/\AA^{2}$)'
        plt.xlabel(xlabel)
332     plt.ylabel(ylabel)
        plt.title(r'$\sigma_{yy}$ vs N')
334     fig.savefig(SyyN,dpi = 300, bbox_inches ='tight')

336     # Sxy
        fig= plt.figure()
338     plt.plot(N,SXY,'rx',linewidth = 0.3,markersize = 1,label = r'$\lambda =
        0.5$')
        plt.plot(N,SXY1,'gx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 1
        $')
340     plt.plot(N,SXY2,'bx',linewidth = 0.3,markersize = 1,label = r'$\lambda =
        1.5$')
        plt.plot(N,SXY3,'cx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 2
        $')
342
        leg = plt.legend(prop={'size':10})
344     leg.get_frame().set_alpha(0.5)
        xlabel = r'Iterations'
346     ylabel = r'$\sigma_{xy}$ ($eV/\AA^{2}$)'
        plt.xlabel(xlabel)
348     plt.ylabel(ylabel)
        plt.title(r'$\sigma_{xy}$ vs N')
350     fig.savefig(SxyN,dpi = 300, bbox_inches ='tight')

352     # P
        fig= plt.figure()
354     plt.plot(N,P,'rx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 0.5$
        ')
        plt.plot(N,P1,'gx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 1$'
        )
356     plt.plot(N,P2,'bx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 1.5
        $')
        plt.plot(N,P3,'cx',linewidth = 0.3,markersize = 1,label = r'$\lambda = 2$'
        )
358
        leg = plt.legend(prop={'size':10})
360     leg.get_frame().set_alpha(0.5)
        xlabel = r'Iterations'
362     ylabel = r'Hydrostatic pressure ($eV/\AA^{2}$)'
        plt.xlabel(xlabel)
364     plt.ylabel(ylabel)
        plt.title(r'Hydrostatic pressure vs N')
366     fig.savefig(PN,dpi = 300, bbox_inches ='tight')


368
        #--------------------plot F_max vs N -------------------------
370     fig= plt.figure()
        plt.plot(N,F,'r-',linewidth = 0.6,markersize = 1,label = r'$\lambda = 0.5$
        ')
372     plt.plot(N,F1,'g-',linewidth = 0.6,markersize = 1,label = r'$\lambda = 1$'
        )
        plt.plot(N,F2,'b-',linewidth = 0.6,markersize = 1,label = r'$\lambda = 1.5
```

```
    $')
374     plt.plot(N,F3,'c-',linewidth = 0.6,markersize = 1,label = r'$\lambda = 2$'
        )

376     leg = plt.legend(prop={'size':10})
        leg.get_frame().set_alpha(0.5)
378     xlabel = r'Iterations'
        ylabel = r'Maximum Force ($eV/\AA$)'
380     plt.xlabel(xlabel)
        plt.ylabel(ylabel)
382     plt.title(r'Maximum Force vs N')
        fig.savefig(FN,dpi = 300, bbox_inches ='tight')
384

386     #--------------------plot F_8 vs N ---------------------------
        fig= plt.figure()
388     plt.plot(N,F4,'r-',linewidth = 0.6,markersize = 1,label = r'$\lambda = 8$'
        )
        leg = plt.legend(prop={'size':10})
390     leg.get_frame().set_alpha(0.5)
        xlabel = r'Iterations'
392     ylabel = r'Maximum Force ($eV/\AA$)'
        plt.ylim ([0,0.14])
394     plt.xlabel(xlabel)
        plt.ylabel(ylabel)
396     plt.title(r'Maximum Force vs N')
        fig.savefig(F8,dpi = 300, bbox_inches ='tight')
398

400     #--------------------plot E_8 vs N ---------------------------
        fig = plt.figure()
402     plt.plot(N,E4,'r-',linewidth = 0.6,markersize = 1,label = r'$\lambda = 8$'
        )
        leg = plt.legend(prop={'size':10})
404     leg.get_frame().set_alpha(0.5)
        xlabel = r'Iternations'
406     ylabel = r'Potential Energy (eV)'
        plt.ylim ([-12,12])
408     plt.xlabel(xlabel)
        plt.ylabel(ylabel)
410     plt.title(r'Potential Energy vs N')
        fig.savefig(E8,dpi = 300, bbox_inches ='tight')
412

414 def main():
        #-------------------------------------------------------------
416     #plot the initial configuration and rdf
        #-------------------------------------------------------------
418     plotconfig("config.txt",'initial_config.png')
        plotrdf("rdf.txt","initial_rdf.png")
420
        #-------------------------------------------------------------
422     #plot the final configuration and rdf of the 4 experiments
        #with different lambdas
```

```
    #-------------------------------------------------------------

    #---------------lambda = 0.5--------------------
    plotconfig("l=05_config.txt","l=05_config.png")
    plotrdf("l=05_rdf.txt","l=05_rdf.png")

    #---------------lambda = 1 --------------------
    plotconfig("l=1_config.txt","l=1_config.png")
    plotrdf("l=1_rdf.txt","l=1_rdf.png")

    #---------------lambda = 1.5--------------------
    plotconfig("l=15_config.txt","l=15_config.png")
    plotrdf("l=15_rdf.txt","l=15_rdf.png")

    #---------------lambda = 2 --------------------
    plotconfig("l=2_config.txt","l=2_config.png")
    plotrdf("l=2_rdf.txt","l=2_rdf.png")

    #---------------lambda = 8 --------------------
    plotconfig("l=8_config.txt","l=8_config.png")
    plotrdf("l=8_rdf.txt","l=8_rdf.png")

    #-----------------------------------------------
    #study the evolution lambda = 1.5
    #-----------------------------------------------

    #iteration 250
    plotconfig("250config.txt","250config.png")
    plotrdf("250rdf.txt","250rdf.png")

    #iteration 500
    plotconfig("500config.txt","500config.png")
    plotrdf("500rdf.txt","500rdf.png")

    #iteration 1000
    plotconfig("1000config.txt","1000config.png")
    plotrdf("1000rdf.txt","1000rdf.png")

    #iteration 2000
    plotconfig("2000config.txt","2000config.png")
    plotrdf("2000rdf.txt","2000rdf.png")

    #-----------------------------------------------
    #plot variables vs N
    #-----------------------------------------------
    plot_vsN("l=0.5_vsN.txt","l=1_vsN.txt","l=1.5_vsN.txt",
            "l=2_vsN.txt","l=8_vsN.txt","E_vs_N.png",
            "Sxx_vs_N.png","Syy_vs_N.png","Sxy_vs_N.png",
            "P_vs_N.png","F_vs_N.png","P8_vs_N.png","F8_vs_N.png")

main()
```

Listing 4: hw3.py