NB: The graded, first version of the report must be returned if you hand in a second time!

H1a: Molecular Dynamics simulation - static properties

Victor Nilsson and Simon Nilsson

November 22, 2016

Task №	Points	Avail. points
Σ		

Introduction

Molecular dynamics is a simulation of the movement of atoms and molecules. What is of interest in such a simulations is e.g. the trajectories of the atoms given specific surrounding parameters such as temperature, pressure, crystal formation etc. For this homeproblem we study the dynamics of aluminium atoms in a FCC crystal lattice.

For the homeproblems we simulate the dynamics of the aluminium atoms by simulating their interactions. At the start of each simulation the atoms are placed on a FCC lattice with lattice parameter 4.046 . These atoms are initially displaced about 5 % of the lattice parameter and their velocities are set to zero.

Problem 1

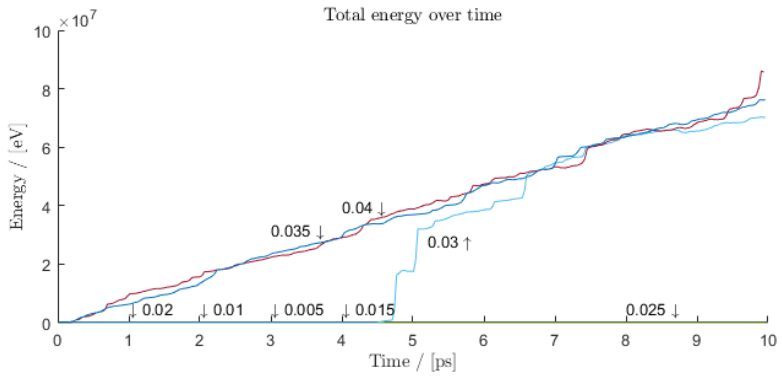

Figure 1: For different time-steps the energy evolves differently over time, in the figure we there are four simulations with different time-steps. The time-steps increases with 0.01 ps for each simulation and the total energy starts to increase for time-steps of 0.03 ps.

For the simulations we updated the positions and velocities of the particles by applying the velocity verlet algorithm.

- $v \leftarrow v + \frac{1}{2}a\Delta t$, update velocity from current acceleration.
- $q \leftarrow q + v\Delta t$, update displacement from current velocity.
- $a \leftarrow a$, update acceleration from the applied forces.
- $v \leftarrow v + \frac{1}{2}a\Delta t$, update velocity from new acceleration.

For this algorithm to be stable, we need to choose a time-step that conserves the total energy. With the algorithm implemented we can simulate the system and study the time evolution of the total energies. In Fig. 1 we can see the implications of different time-steps. We can see that the energy is unstable for $\Delta t = 0.03$ ps and stable for the previous one at $\Delta t = 0.025$ ps. To have some safety margin we will, for the rest of the report, use a time-step of $\Delta t = 0.005$ ps.

Problem 3

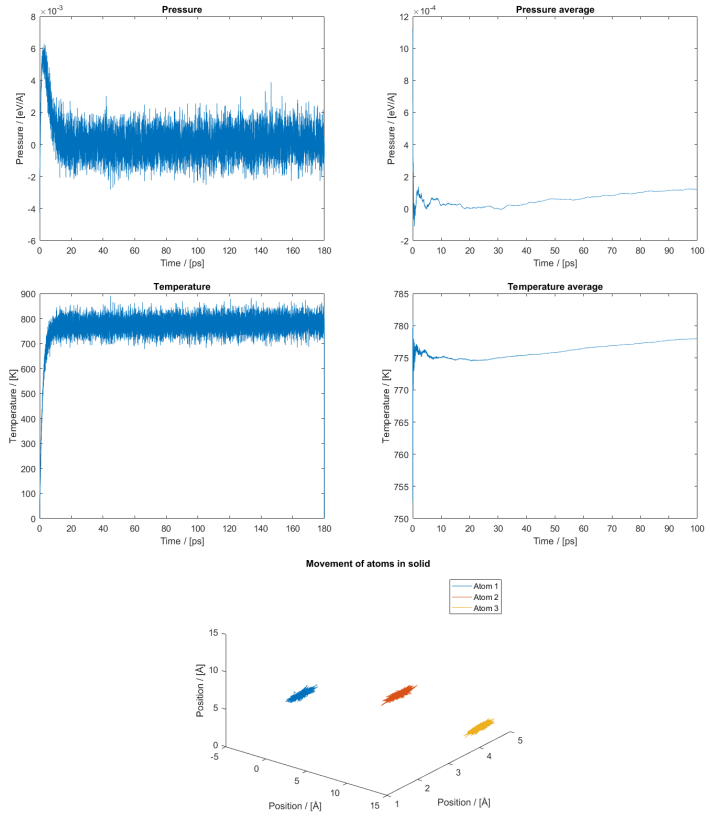

Figure 2: After the equilibrium, both the pressure and the temperature needs some time to stabilize after rescaling the velocities, but remains stable for longer time. The equalisation temperature was set to 500 C°.

In order to set the system to a certain temperature, a technique involving scaling all the velocities during a equilibrating state. Since the temperature depends on the kinetic energies which in turn depends on the velocities, the temperature can thusly be changed by changing velocities. In figure 3 we can see the temperature after setting the temperature to 500 C°. There are some fluctuations in the beginning due to the rescaling of the velocities.

Problem 4

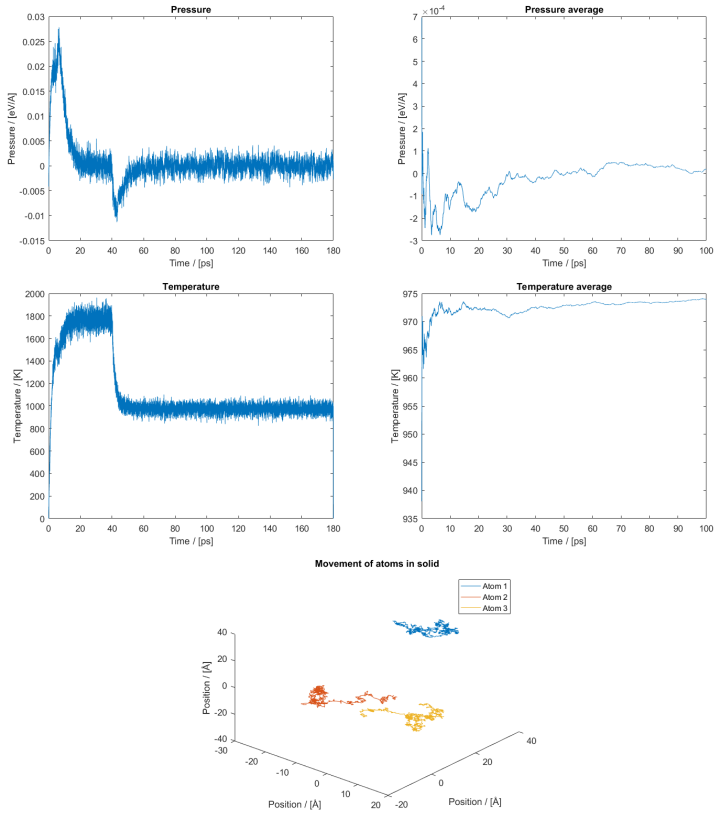

Figure 3: After the equilibrium, both the pressure and the temperature needs some time to stabilize after rescaling the velocities, but remains stable for longer time. The equalisation temperature was first set to 1000 C° for the smelting and then the temperature was reduced to 700 C°.

Problem 5

$1\ \frac{\mathrm{j}}{\mathrm{molK}} = 1.0366e-05\ \frac{\mathrm{eV}}{\mathrm{K}}$
$C_v\ [AL] : 24.20\ \frac{\mathrm{j}}{\mathrm{molK}} = 1.0366e-05\ \frac{\mathrm{eV}}{\mathrm{K}}$
From our MD simulations we obtained the following values for C_V when using the potential and kinetic energy fluctuations respectively.

Table 1: Heat capacity obtained by measuring energy fluctuations.

Temperature	**500° C**	**700° C**
$C_V/(\mathrm{eV/kg\,K})$ (kinetic)	$5.819521 \cdot 10^{-2}$	$5.056438 \cdot 10^{-2}$
$C_V/(\mathrm{eV/kg\,K})$ (potential)	$5.836188 \cdot 10^{-2}$	$5.056541 \cdot 10^{-2}$

Problem 6

When instead using the relation

$$C_V = \left(\frac{\partial E}{\partial T}\right)_{N,V} \tag{1}$$

to compute the heat capacity, and approximate it with a difference quota, we obtain the results found in the table below.

If we compare these results to those from the previous problem we see that they are slightly larger and the result for 700 degrees deviates by a larger margin. It's possible

Table 2: Heat capacity obtained by approximating the partial energy derivative with respect to the temperature.

Temperature	**500° C**	**700° C**
$C_V/(\text{eV/kg K})$	$6.436 \cdot 10^{-2}$	$8.131 \cdot 10^{-2}$

that a longer equilibration time would yield a more stable temperature than was obtained now and therefore a more accurate heat capacity. A longer measuring time would also increase the accuracy of the result, as well as doing more simulation and averaging the different results. A ΔT of 5° C was used here, but further experiementing with this parameter could yield a better result as well.

Problem 7

The radial distribution function obtained can be found in figure 4. Using Matlab we find that the first peak is at the distance 2.85 Å, which corresponds to the shortest distance in a fcc structure with the unit cell length of 4.046 Å. This is the distance between one of the corner atoms and a face centered atom close to that corner, which is expected. The other larger peaks are around 5 Å and 7.17 Å. These align with the two next shortest distances in an fcc structure.

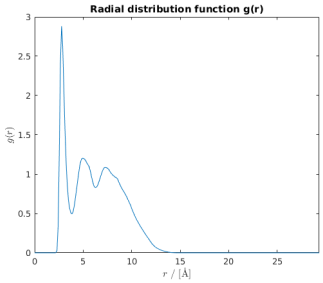

Figure 4: The radial function is computed by taking the histogram over all the internal distances between the atoms then divided by the random distribution of the same density.

Problem 8

Below (Fig. 5) are two approximations of the static structure function. The leftmost figure is obtained by integrating the result from Problem 7 and the rightmost one is obtained via simulation.

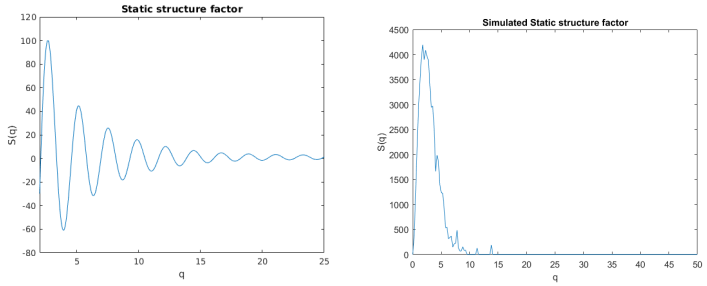

Figure 5: The static structure function computed both from the radial distribution function in figure 4 in the left and using bins in Fourier space to the right.

Concluding discussion

A Source code

A.1 Task1/MD_main.c

```
/*
 MD_main.c

 Created by Anders Lindman on 2013-10-31.
 */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "initfcc.h"
#include "alpotential.h"
#define nbr_of_particles 256
#define nbr_of_dimensions 3

/* Main program */
int main()
{
    srand(time(NULL));

    /* Simulation parameters */
    double m_AL; // Mass of atom
    double cell_length; // Side length of supercell

    double lattice_spacing; // Smallest length between atoms
    double initial_displacement;    // Initial displacement of the atoms from ↩
        their
                                    // lattice positions

    double lattice_param;   // Lattice parameter, length of each side in the
                            // unit cell
    double timesteps[8];

    FILE *file;

    int time_length = 10;

    /* Initialize parameters*/
    initial_displacement = 0.05;
    lattice_param = 4.046; // For aluminium (  )
    lattice_spacing = lattice_param/sqrt(2.0);
    //timestep = 0.01; // 0.1 Bad, 0.01 Seems decent

    m_AL = 0.0027964; // In ASU
    cell_length = 4*lattice_param;  // Side of the supercell: The 256 atoms are
                                    // structured in a block of 4x4x4 unit cells

    // Test different timestep with 0.01    difference
    for (int i = 0; i < 8; i++)
        timesteps[i]=0.005*(i+1);

    for (int t = 0; t < 8; t++)
    {
        // Current timestep and number of timesteps
        double timestep = timesteps[t];
        int nbr_of_timesteps = (int)(time_length/timestep);

        /* Current displacement, velocities, and acceleratons */
        double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
        double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
        double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces



        /* Allocate memory for large vectors */
        /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional ↩
            array*/
        double* energy_pot =(double*)malloc(nbr_of_timesteps*sizeof(double));
        double* energy_kin = (double*)malloc(nbr_of_timesteps*sizeof(double));

        /* Put atoms on lattice */
        init_fcc(q, 4, lattice_param);

        /* Initial conditions */

        for (int i = 0; i < nbr_of_particles; i++){
            for (int j = 0; j < nbr_of_dimensions; j++){

                // Initial perturbation from equilibrium
```

```
                q[i][j] +=lattice_spacing* initial_displacement
                    * ((double)rand()/(double)RAND_MAX);
            }
        }
        energy_pot[0]=get_energy_AL(q,cell_length,nbr_of_particles);
        energy_kin[0]=get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,m_AL);

        get_forces_AL(f,q,cell_length,nbr_of_particles);


        /* Simulation */
        for (int i = 1; i < nbr_of_timesteps; i++)
        {
            /** Verlet algorithm **/
            /* Half step for velocity */
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
                }
            }

            /* Update displacement*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    q[j][k] += timestep * v[j][k];
                }
            }

            /* Forces */
            get_forces_AL(f,q,cell_length,nbr_of_particles);

            /* Final velocity*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
                }
            }

            /* Calculate energy */
            // Potential energy
            energy_pot[i] = get_energy_AL(q,cell_length,nbr_of_particles);
            // Kinetic energy
            energy_kin[i] = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,↩
                m_AL);
        }
        char str[80];
        char S[3];
        sprintf(S, "%.3f", timestep);

          strcpy (str,"data/energy");
          strcat (str,S);
          strcat (str,".dat");


        /* Save energies to file */
        file = fopen(str,"w");

        double current_time;
        for (int i = 0; i < nbr_of_timesteps; i ++)
        {
            current_time = i*timestep;
            fprintf(file, "%.4f \t", current_time);
            fprintf(file, "%.4f \t", energy_pot[i]);
            fprintf(file, "%.4f \n", energy_kin[i]);
        }
        fclose(file);



        free(energy_kin); energy_kin=NULL;
        free(energy_pot); energy_pot=NULL;
    }

    return 0;
}
```

A.2 Task3/MD_main.c

```

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "initfcc.h"
#include "alpotential.h"
```

```c
#define nbr_of_particles 256
#define nbr_of_timesteps 1e4
#define nbr_of_timesteps_eq 4000
#define nbr_of_dimensions 3

double boundary_condition(double,double);



/* Main program */
int main()
{
    srand(time(NULL));

    /* Simulation parameters */
    double m_AL; // Mass of atom
    double cell_length; // Side length of supercell
    double volume;
    double lattice_spacing; // Smallest length between atoms
    double initial_displacement;    // Initial displacement of the atoms from their
                                    // lattice positions
    double lattice_param;   // Lattice parameter, length of each side in the
                            // unit cell
    double timestep;
    double temperature_eq[] = { 500.0+273.15, 500.0+273.15 };
    double pressure_eq = 101325e-11/1.602; // 1 atm in ASU

    FILE *file;


    /* Current displacement, velocities, and acceleratons */
    double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
    double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
    double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces

    /* Allocate memory for large vectors */
    /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional array*/
    #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+nbr_of_dimensions*j+k])
    double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*nbr_of_dimensions*sizeof(double));

    double* energy          = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* energy_kin      = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* virial          = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* pressure_avg    = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* temperature     = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));
    double* pressure        = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));

    //TODO go over parameters again
    /* Initialize parameters*/
    initial_displacement    = 0.05;
    lattice_param           = 4.046; // For aluminium (  )
    lattice_spacing         = lattice_param/sqrt(2.0);
    timestep                = 0.01; // 0.1 Bad, 0.01 Seems decent
    m_AL                    = 0.0027964; // In ASU
    cell_length             = 4*lattice_param;  // Side of the supercell: The 256 atoms are
                                                // structured in a block of 4x4x4 unit cells
    volume                  = pow(cell_length, 3);

    // Initialize all displacements, for all times, as 0
    for (int i  = 0; i < nbr_of_timesteps; i++){
        for (int j = 0; j < nbr_of_particles; j++){
            for (int k = 0; k < nbr_of_dimensions; k++){
                qq(i,j,k) = 0;
            }
        }
    }

    /* Put atoms on lattice */
    init_fcc(q, 4, lattice_param);


    /* Initial conditions */
    for (int i = 0; i < nbr_of_particles; i++){
        for (int j = 0; j < nbr_of_dimensions; j++){

            // Initial perturbation from equilibrium
            q[i][j] += lattice_spacing * initial_displacement
```

```
87                  * ((double)rand()/(double)RAND_MAX);
88
89          }
90      }
91
92
93      get_forces_AL(f, q, cell_length, nbr_of_particles);
94
95      /* Simulation */
96      /* Equilibrium stage */
97
98      double inst_temperature_eq;
99      double inst_pressure_eq;
100     double alpha_T = 1.0;
101     double alpha_P = 1.0;
102     double energy_kin_eq = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,↩
            m_AL);
103     double virial_eq = get_virial_AL(q,cell_length,nbr_of_particles);
104
105     temperature[0]  = instantaneous_temperature(energy_kin_eq, nbr_of_particles)↩
            ;
106     pressure[0]     = instantaneous_pressure(virial_eq, temperature[0], ↩
            nbr_of_particles, volume);
107
108     for (int equil = 0; equil < 2; equil++) {
109         for (int i = 1; i < nbr_of_timesteps_eq; i++)
110         {
111             /** Verlet algorithm **/
112             /* Half step for velocity */
113             for (int j = 0; j < nbr_of_particles; j++){
114                 for (int k = 0; k < nbr_of_dimensions; k++){
115                     v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
116                 }
117             }
118
119             /* Update displacement*/
120             for (int j = 0; j < nbr_of_particles; j++){
121                 for (int k = 0; k < nbr_of_dimensions; k++){
122                     q[j][k] += timestep * v[j][k];
123                 }
124             }
125
126             /* Forces */
127             get_forces_AL(f,q,cell_length,nbr_of_particles);
128
129             /* Final velocity*/
130             for (int j = 0; j < nbr_of_particles; j++){
131                 for (int k = 0; k < nbr_of_dimensions; k++){
132                     v[j][k] += timestep * 0.5* f[j][k]/m_AL;
133                 }
134             }
135
136             /* Calculate energy */
137             // Kinetic energy
138             energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, ↩
                    nbr_of_particles, m_AL);
139
140             virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);
141
142
143             inst_temperature_eq = instantaneous_temperature(energy_kin_eq, ↩
                    nbr_of_particles);
144             temperature[equil*(nbr_of_timesteps_eq-1) + i] = inst_temperature_eq↩
                    ;
145             inst_pressure_eq = instantaneous_pressure(virial_eq, ↩
                    inst_temperature_eq,
146                 nbr_of_particles, volume);
147             pressure[equil*(nbr_of_timesteps_eq-1) + i] = inst_pressure_eq;
148
149
150             // Update alhpas
151             alpha_T = 1.0 + 0.01*(temperature_eq[equil]-inst_temperature_eq)/↩
                    inst_temperature_eq;
152             alpha_P = 1.0 - 0.01*(pressure_eq - inst_pressure_eq);
153
154
155             // Scale velocities
156             for (int j = 0; j < nbr_of_particles; j++){
157                 for (int k = 0; k < nbr_of_dimensions; k++){
158                     v[j][k] *= sqrt(alpha_T);
159                 }
160             }
161
162             // Scale positions and volume
163             cell_length *= pow(alpha_P, 1.0/3.0);
164             volume = pow(cell_length, 3);
165             for (int j = 0; j < nbr_of_particles; j++) {
166                 for (int k = 0; k < nbr_of_dimensions; k++) {
167                     q[j][k] *= pow(alpha_P, 1.0/3.0);
168                 }
169             }
```

```
170 
171         }
172     }
173 
174     printf("Equilibration done.\n");
175     printf("Cell length: %.8f \n", cell_length);
176 
177     for (int i = 0; i < nbr_of_particles; i++){
178         for (int j = 0; j < nbr_of_dimensions; j++){
179             qq(0,i,j)=q[i][j];
180         }
181     }
182 
183     // Compute energies, temperature etc. at equilibrium
184     energy[0] = get_energy_AL(q, cell_length, nbr_of_particles);
185     virial[0] = get_virial_AL(q, cell_length, nbr_of_particles);
186     energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL)↩
          ;
187     temperature_avg[0] = instantaneous_temperature(energy_kin[0], ↩
          nbr_of_particles);
188     pressure_avg[0] = instantaneous_pressure(virial[0], temperature_avg[0],
          nbr_of_particles, volume);
189 
190     /* Simulation after equilibrium*/
191     for (int i = 1; i < nbr_of_timesteps; i++)
192     {
193         /** Verlet algorithm **/
194         /* Half step for velocity */
195         for (int j = 0; j < nbr_of_particles; j++){
196             for (int k = 0; k < nbr_of_dimensions; k++){
197                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
198             }
199         }
200 
201         /* Update displacement*/
202         for (int j = 0; j < nbr_of_particles; j++){
203             for (int k = 0; k < nbr_of_dimensions; k++){
204                 q[j][k] += timestep * v[j][k];
205             }
206         }
207 
208         /* Update Forces */
209         get_forces_AL(f, q, cell_length, nbr_of_particles);
210 
211         /* Final velocity*/
212         for (int j = 0; j < nbr_of_particles; j++){
213             for (int k = 0; k < nbr_of_dimensions; k++){
214                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
215             }
216         }
217 
218         /* Calculate energy */
219         // Potential energy
220         energy[i] = get_energy_AL(q, cell_length, nbr_of_particles);
221         // Kinetic energy
222         energy_kin[i] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, ↩
              m_AL);
223 
224         virial[i] = get_virial_AL(q, cell_length, nbr_of_particles);
225 
226         // Temperature
227         temperature_avg[i] = averaged_temperature(energy_kin, nbr_of_particles, ↩
              i);
228         temperature[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_temperature(↩
              energy_kin[i],
229             nbr_of_particles);
230 
231 
232         // Pressure
233         pressure_avg[i] = averaged_pressure(virial, energy_kin, volume, i);
234         pressure[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_pressure(virial[↩
              i],
235             temperature[2*(nbr_of_timesteps_eq-1) + i],
236             nbr_of_particles, volume);
237 
238 
239         /* Save current displacements to array*/
240         for (int j = 0; j < nbr_of_particles; j++){
241             for (int k = 0; k < nbr_of_dimensions; k++){
242                 qq(i,j,k)=q[j][k];
243             }
244         }
245 
246     } // equilibration/simulation
247 
248     /* Save data to file*/
249     file = fopen("displacement.dat","w");
250 
251     double current_time;
252     for (int i = 0; i < nbr_of_timesteps; i ++) {
253         current_time = i*timestep;
254 
```

```
        fprintf(file, "%.4f \t", current_time );
        for (int j = 0; j < nbr_of_particles; j++) {
            for (int k = 0; k < nbr_of_dimensions; k++) {
                fprintf(file, "%.4f \t", qq(i,j,k));
            }
        }
        fprintf(file, "\n");
    }
    fclose(file);

    /* Save energies to file */
    file = fopen("energy.dat","w");

    for (int i = 0; i < nbr_of_timesteps; i ++) {
        current_time = i*timestep;
        fprintf(file, "%.4f \t", current_time);
        fprintf(file, "%.4f \t", energy[i]);
        fprintf(file, "%.4f \n", energy_kin[i]);
    }
    fclose(file);

    // Save temperature to file
    file = fopen("temperature.dat", "w");
    for (int i = 0; i < 2*nbr_of_timesteps_eq+nbr_of_timesteps; i++) {
        current_time = i*timestep;
        fprintf(file, "%.3f \t %e\n", current_time, temperature[i]);
    }
    fclose(file);

    file = fopen("temperature_avg.dat", "w");
    for (int i = 0; i < nbr_of_timesteps; i++) {
        current_time = i*timestep;
        fprintf(file, "%.3f \t %e\n", current_time, temperature_avg[i]);
    }
    fclose(file);

    // Save pressure to file
    file = fopen("pressure.dat", "w");
    for (int i = 0; i < 2*nbr_of_timesteps_eq+nbr_of_timesteps; i++) {
        current_time = i*timestep;
        fprintf(file, "%.3f \t%e \n", current_time, pressure[i]);
    }
    fclose(file);

    file = fopen("pressure_avg.dat", "w");
    for (int i = 0; i < nbr_of_timesteps; i++) {
        current_time = i*timestep;
        fprintf(file, "%.3f \t %e\n", current_time, pressure_avg[i]);
    }
    fclose(file);


    free(energy_kin);       energy_kin = NULL;
    free(energy);           energy = NULL;
    free(disp_arr);         disp_arr = NULL;
    free(virial);           virial = NULL;
    free(temperature_avg);  temperature_avg = NULL;
    free(pressure_avg);     pressure_avg = NULL;
    free(temperature);      temperature = NULL;
    free(pressure);         pressure = NULL;

    return 0;
}
```

A.3 Task4/MD_main.c

```

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "initfcc.h"
#include "alpotential.h"
#define nbr_of_particles 256
#define nbr_of_timesteps 1e4
#define nbr_of_timesteps_eq 4000
#define nbr_of_dimensions 3

double boundary_condition(double,double);



/* Main program */
int main()
{
```

```
    srand(time(NULL));

    /* Simulation parameters */
    double m_AL; // Mass of atom
    double cell_length; // Side length of supercell
    double volume;
    double lattice_spacing; // Smallest length between atoms
    double initial_displacement;    // Initial displacement of the atoms from their
                                    // lattice positions
    double lattice_param;   // Lattice parameter, length of each side in the
                            // unit cell
    double timestep;
    double temperature_eq[] = { 1500.0+273.15, 700.0+273.15 };
    double pressure_eq = 101325e-11/1.602; // 1 atm in ASU

    FILE *file;


    /* Current displacement, velocities, and acceleratons */
    double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
    double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
    double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces

    /* Allocate memory for large vectors */
    /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional array*/
    #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+nbr_of_dimensions*j+k])
    double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*nbr_of_dimensions*sizeof(double));

    double* energy          = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* energy_kin      = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* virial          = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* pressure_avg    = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* temperature     = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));
    double* pressure        = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));

    //TODO go over parameters again
    /* Initialize parameters*/
    initial_displacement    = 0.05;
    lattice_param           = 4.046; // For aluminium (  )
    lattice_spacing         = lattice_param/sqrt(2.0);
    timestep                = 0.01; // 0.1 Bad, 0.01 Seems decent
    m_AL                    = 0.0027964; // In ASU
    cell_length             = 4*lattice_param;  // Side of the supercell: The 256 atoms are
                                                // structured in a block of 4x4x4 unit cells
    volume                  = pow(cell_length, 3);

    // Initialize all displacements, for all times, as 0
    for (int i  = 0; i < nbr_of_timesteps; i++){
        for (int j = 0; j < nbr_of_particles; j++){
            for (int k = 0; k < nbr_of_dimensions; k++){
                qq(i,j,k) = 0;
            }
        }
    }

    /* Put atoms on lattice */
    init_fcc(q, 4, lattice_param);


    /* Initial conditions */
    for (int i = 0; i < nbr_of_particles; i++){
        for (int j = 0; j < nbr_of_dimensions; j++){

            // Initial perturbation from equilibrium
            q[i][j] += lattice_spacing * initial_displacement
                * ((double)rand()/(double)RAND_MAX);

        }
    }


    get_forces_AL(f, q, cell_length, nbr_of_particles);

    /* Simulation */
    /* Equilibrium stage */

    double inst_temperature_eq;
```

```
    double inst_pressure_eq;
    double alpha_T = 1.0;
    double alpha_P = 1.0;
    double energy_kin_eq = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,m_AL);
    double virial_eq = get_virial_AL(q,cell_length,nbr_of_particles);

    temperature[0]  = instantaneous_temperature(energy_kin_eq, nbr_of_particles);
    pressure[0]     = instantaneous_pressure(virial_eq, temperature[0], nbr_of_particles, volume);

    for (int equil = 0; equil < 2; equil++) {
        for (int i = 1; i < nbr_of_timesteps_eq; i++)
        {
            /** Verlet algorithm **/
            /* Half step for velocity */
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
                }
            }

            /* Update displacement*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    q[j][k] += timestep * v[j][k];
                }
            }

            /* Forces */
            get_forces_AL(f,q,cell_length,nbr_of_particles);

            /* Final velocity*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5* f[j][k]/m_AL;
                }
            }

            /* Calculate energy */
            // Kinetic energy
            energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL);

            virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);


            inst_temperature_eq = instantaneous_temperature(energy_kin_eq, nbr_of_particles);
            temperature[equil*(nbr_of_timesteps_eq-1) + i] = inst_temperature_eq;
            inst_pressure_eq = instantaneous_pressure(virial_eq, inst_temperature_eq,
                nbr_of_particles, volume);
            pressure[equil*(nbr_of_timesteps_eq-1) + i] = inst_pressure_eq;


            // Update alhpas
            alpha_T = 1.0 + 0.01*(temperature_eq[equil]-inst_temperature_eq)/inst_temperature_eq;
            alpha_P = 1.0 - 0.01*(pressure_eq - inst_pressure_eq);


            // Scale velocities
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] *= sqrt(alpha_T);
                }
            }

            // Scale positions and volume
            cell_length *= pow(alpha_P, 1.0/3.0);
            volume = pow(cell_length, 3);
            for (int j = 0; j < nbr_of_particles; j++) {
                for (int k = 0; k < nbr_of_dimensions; k++) {
                    q[j][k] *= pow(alpha_P, 1.0/3.0);
                }
            }

        }
    }

    printf("Equilibration done.\n");
    printf("Cell length: %.8f \n", cell_length);

    for (int i = 0; i < nbr_of_particles; i++){
        for (int j = 0; j < nbr_of_dimensions; j++){
            qq(0,i,j)=q[i][j];
        }
    }
```

```
182
183     // Compute energies, temperature etc. at equilibrium
184     energy[0] = get_energy_AL(q, cell_length, nbr_of_particles);
185     virial[0] = get_virial_AL(q, cell_length, nbr_of_particles);
186     energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL)↩
            ;
187     temperature_avg[0] = instantaneous_temperature(energy_kin[0], ↩
            nbr_of_particles);
188     pressure_avg[0] = instantaneous_pressure(virial[0], temperature_avg[0],
189         nbr_of_particles, volume);
190
191     /* Simulation after equilibrium*/
192     for (int i = 1; i < nbr_of_timesteps; i++)
193     {
194         /** Verlet algorithm **/
195         /* Half step for velocity */
196         for (int j = 0; j < nbr_of_particles; j++){
197             for (int k = 0; k < nbr_of_dimensions; k++){
198                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
199             }
200         }
201
202         /* Update displacement*/
203         for (int j = 0; j < nbr_of_particles; j++){
204             for (int k = 0; k < nbr_of_dimensions; k++){
205                 q[j][k] += timestep * v[j][k];
206             }
207         }
208
209         /* Update Forces */
210         get_forces_AL(f, q, cell_length, nbr_of_particles);
211
212         /* Final velocity*/
213         for (int j = 0; j < nbr_of_particles; j++){
214             for (int k = 0; k < nbr_of_dimensions; k++){
215                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
216             }
217         }
218
219         /* Calculate energy */
220         // Potential energy
221         energy[i] = get_energy_AL(q, cell_length, nbr_of_particles);
222         // Kinetic energy
223         energy_kin[i] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, ↩
                m_AL);
224
225         virial[i] = get_virial_AL(q, cell_length, nbr_of_particles);
226
227         // Temperature
228         temperature_avg[i] = averaged_temperature(energy_kin, nbr_of_particles, ↩
                i);
229         temperature[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_temperature(↩
                energy_kin[i],
230             nbr_of_particles);
231
232
233         // Pressure
234         pressure_avg[i] = averaged_pressure(virial, energy_kin, volume, i);
235         pressure[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_pressure(virial[↩
                i],
236             temperature[2*(nbr_of_timesteps_eq-1) + i],
237             nbr_of_particles, volume);
238
239
240         /* Save current displacements to array*/
241         for (int j = 0; j < nbr_of_particles; j++){
242             for (int k = 0; k < nbr_of_dimensions; k++){
243                 qq(i,j,k)=q[j][k];
244             }
245         }
246
247     } // equilibration/simulation
248
249     /* Save data to file*/
250     file = fopen("displacement.dat","w");
251
252     double current_time;
253     for (int i = 0; i < nbr_of_timesteps; i ++) {
254         current_time = i*timestep;
255         fprintf(file, "%.4f \t", current_time );
256         for (int j = 0; j < nbr_of_particles; j++) {
257             for (int k = 0; k < nbr_of_dimensions; k++) {
258                 fprintf(file, "%.4f \t", qq(i,j,k));
259             }
260         }
261         fprintf(file, "\n");
262     }
263     fclose(file);
264
265     /* Save energies to file */
266     file = fopen("energy.dat","w");
```

```
267
268     for (int i = 0; i < nbr_of_timesteps; i ++) {
269         current_time = i*timestep;
270         fprintf(file, "%.4f \t", current_time);
271         fprintf(file, "%.4f \t", energy[i]);
272         fprintf(file, "%.4f \n", energy_kin[i]);
273     }
274     fclose(file);
275
276     // Save temperature to file
277     file = fopen("temperature.dat", "w");
278     for (int i = 0; i < 2*nbr_of_timesteps_eq+nbr_of_timesteps; i++) {
279         current_time = i*timestep;
280         fprintf(file, "%.3f \t %e\n", current_time, temperature[i]);
281     }
282     fclose(file);
283
284     file = fopen("temperature_avg.dat", "w");
285     for (int i = 0; i < nbr_of_timesteps; i++) {
286         current_time = i*timestep;
287         fprintf(file, "%.3f \t %e\n", current_time, temperature_avg[i]);
288     }
289     fclose(file);
290
291     // Save pressure to file
292     file = fopen("pressure.dat", "w");
293     for (int i = 0; i < 2*nbr_of_timesteps_eq+nbr_of_timesteps; i++) {
294         current_time = i*timestep;
295         fprintf(file, "%.3f \t%e \n", current_time, pressure[i]);
296     }
297     fclose(file);
298
299     file = fopen("pressure_avg.dat", "w");
300     for (int i = 0; i < nbr_of_timesteps; i++) {
301         current_time = i*timestep;
302         fprintf(file, "%.3f \t %e\n", current_time, pressure_avg[i]);
303     }
304     fclose(file);
305
306
307     free(energy_kin);       energy_kin = NULL;
308     free(energy);           energy = NULL;
309     free(disp_arr);         disp_arr = NULL;
310     free(virial);           virial = NULL;
311     free(temperature_avg);  temperature_avg = NULL;
312     free(pressure_avg);     pressure_avg = NULL;
313     free(temperature);      temperature = NULL;
314     free(pressure);         pressure = NULL;
315
316     return 0;
317 }
```

A.4 Task5/MD_main.c

```
1  /*
2   MD_main.c
3
4   Created by Anders Lindman on 2013-10-31.
5   */
6
7  #include <stdio.h>
8  #include <math.h>
9  #include <stdlib.h>
10 #include <time.h>
11 #include "initfcc.h"
12 #include "alpotential.h"
13 #define nbr_of_particles 256
14 #define nbr_of_timesteps 1e4
15 #define nbr_of_timesteps_eq 4000
16 #define nbr_of_dimensions 3
17
18 double boundary_condition(double,double);
19
20
21
22 /* Main program */
23 int main()
24 {
25     srand(time(NULL));
26
27     /* Simulation parameters */
28     double m_AL; // Mass of atom
29     double cell_length; // Side length of supercell
30     double volume;
31     double lattice_spacing; // Smallest length between atoms
```

```
    double initial_displacement;    // Initial displacement of the atoms from their
                                    // lattice positions
    double lattice_param;   // Lattice parameter, length of each side in the
                            // unit cell
    double timestep;
    double temperature_eq[] = { 1000.0+273.15, 700.0+273.15 };
    double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
    double isothermal_compressibility = 1.0; //0.8645443196; // 1.385e-11 m^2/N = 1.385/1.602    ^3/eV

    FILE *file;


    /* Current displacement, velocities, and acceleratons */
    double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
    double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
    double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces

    double heat_capacity_pot, heat_capacity_kin;

    /* Allocate memory for large vectors */
    /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional array*/
    #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+nbr_of_dimensions*j+k])
    double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*nbr_of_dimensions*sizeof(double));

    double* energy_pot      = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* energy_kin      = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* virial          = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* pressure_avg    = (double*) malloc(nbr_of_timesteps * sizeof(double));
    //double* temperature     = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));
    //double* pressure        = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));

    //TODO go over parameters again
    /* Initialize parameters*/
    initial_displacement    = 0.05;
    lattice_param           = 4.046; // For aluminium (  )
    lattice_spacing         = lattice_param/sqrt(2.0);
    timestep                = 0.001; // 0.1 Bad, 0.01 Seems decent
    m_AL                    = 0.0027964; // In ASU
    cell_length             = 4*lattice_param;  // Side of the supercell: The 256 atoms are
                                                // structured in a block of 4x4x4 unit cells
    volume                  = pow(cell_length, 3);

    // Initialize all displacements, for all times, as 0
    for (int i  = 0; i < nbr_of_timesteps; i++){
        for (int j = 0; j < nbr_of_particles; j++){
            for (int k = 0; k < nbr_of_dimensions; k++){
                qq(i,j,k) = 0;
            }
        }
    }

    /* Put atoms on lattice */
    init_fcc(q, 4, lattice_param);


    /* Initial conditions */
    for (int i = 0; i < nbr_of_particles; i++){
        for (int j = 0; j < nbr_of_dimensions; j++){

            // Initial perturbation from equilibrium
            q[i][j] += lattice_spacing * initial_displacement
                * ((double)rand()/(double)RAND_MAX);

        }
    }


    get_forces_AL(f, q, cell_length, nbr_of_particles);

    /* Simulation */
    /* Equilibrium stage */

    double inst_temperature_eq;
    double inst_pressure_eq;
    double alpha_T = 1.0;
    double alpha_P = 1.0;
```

```
    double energy_kin_eq = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,↩
        m_AL);
    double virial_eq = get_virial_AL(q,cell_length,nbr_of_particles);

    //temperature[0]  = instantaneous_temperature(energy_kin_eq, ↩
        nbr_of_particles);
    //pressure[0]     = instantaneous_pressure(virial_eq, temperature[0], ↩
        nbr_of_particles, volume);

    for (int equil = 0; equil < 2; equil++) {
        for (int i = 1; i < nbr_of_timesteps_eq; i++)
        {
            /** Verlet algorithm **/
            /* Half step for velocity */
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
                }
            }

            /* Update displacement*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    q[j][k] += timestep * v[j][k];
                }
            }

            /* Forces */
            get_forces_AL(f,q,cell_length,nbr_of_particles);

            /* Final velocity*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5* f[j][k]/m_AL;
                }
            }

            /* Calculate energy */
            // Kinetic energy
            energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, ↩
                nbr_of_particles, m_AL);

            virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);


            inst_temperature_eq = instantaneous_temperature(energy_kin_eq, ↩
                nbr_of_particles);
            //temperature[equil*(nbr_of_timesteps_eq-1) + i] = ↩
                inst_temperature_eq;
            inst_pressure_eq = instantaneous_pressure(virial_eq, ↩
                inst_temperature_eq,
                nbr_of_particles, volume);
            //pressure[equil*(nbr_of_timesteps_eq-1) + i] = inst_pressure_eq;


            // Update alhpas
            alpha_T = 1.0 + 0.01*(temperature_eq[equil]-inst_temperature_eq)/↩
                inst_temperature_eq;
            alpha_P = 1.0 - 0.01*isothermal_compressibility*(pressure_eq - ↩
                inst_pressure_eq);

            // DEBUG:alpha
            //printf("%.8f \t %.8f \n", alpha_T, alpha_P);

            // Scale velocities
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] *= sqrt(alpha_T);
                }
            }

            // Scale positions and volume
            cell_length *= pow(alpha_P, 1.0/3.0);
            volume = pow(cell_length, 3);
            for (int j = 0; j < nbr_of_particles; j++) {
                for (int k = 0; k < nbr_of_dimensions; k++) {
                    q[j][k] *= pow(alpha_P, 1.0/3.0);
                }
            }

        }
    }

    printf("Equilibration done.\n");
    printf("Cell length: %.8f \n", cell_length);

    for (int i = 0; i < nbr_of_particles; i++){
        for (int j = 0; j < nbr_of_dimensions; j++){
            qq(0,i,j)=q[i][j];
        }
    }
```

```
191
192     // Compute energies, temperature etc. at equilibrium
193     energy_pot[0] = get_energy_AL(q, cell_length, nbr_of_particles);
194     virial[0] = get_virial_AL(q, cell_length, nbr_of_particles);
195     energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL)
            ;
196     temperature_avg[0] = instantaneous_temperature(energy_kin[0],
            nbr_of_particles);
197     pressure_avg[0] = instantaneous_pressure(virial[0], temperature_avg[0],
198         nbr_of_particles, volume);
199
200     /* Simulation after equilibrium*/
201     for (int i = 1; i < nbr_of_timesteps; i++)
202     {
203         /** Verlet algorithm **/
204         /* Half step for velocity */
205         for (int j = 0; j < nbr_of_particles; j++){
206             for (int k = 0; k < nbr_of_dimensions; k++){
207                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
208             }
209         }
210
211         /* Update displacement*/
212         for (int j = 0; j < nbr_of_particles; j++){
213             for (int k = 0; k < nbr_of_dimensions; k++){
214                 q[j][k] += timestep * v[j][k];
215             }
216         }
217
218         /* Update Forces */
219         get_forces_AL(f, q, cell_length, nbr_of_particles);
220
221         /* Final velocity*/
222         for (int j = 0; j < nbr_of_particles; j++){
223             for (int k = 0; k < nbr_of_dimensions; k++){
224                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
225             }
226         }
227
228         /* Calculate energy */
229         // Potential energy
230         energy_pot[i] = get_energy_AL(q, cell_length, nbr_of_particles);
231         // Kinetic energy
232         energy_kin[i] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles,
                m_AL);
233
234         virial[i] = get_virial_AL(q, cell_length, nbr_of_particles);
235
236         // Temperature
237         temperature_avg[i] = averaged_temperature(energy_kin, nbr_of_particles,
                i);
238         /*temperature[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_temperature
                (energy_kin[i],
239             nbr_of_particles);*/
240
241
242         // Pressure
243         pressure_avg[i] = averaged_pressure(virial, energy_kin, volume, i);
244         /*pressure[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_pressure(
                virial[i],
245             temperature[2*(nbr_of_timesteps_eq-1) + i],
246             nbr_of_particles, volume);*/
247
248
249         /* Save current displacements to array*/
250         for (int j = 0; j < nbr_of_particles; j++){
251             for (int k = 0; k < nbr_of_dimensions; k++){
252                 qq(i,j,k)=q[j][k];
253             }
254         }
255
256     } // equilibration/simulation
257
258     // Compute heat capacity
259     heat_capacity_kin = calculate_heat_capacity_kin(energy_kin, temperature_eq
            [1],
260         nbr_of_particles, nbr_of_timesteps);
261     heat_capacity_pot = calculate_heat_capacity_pot(energy_pot, temperature_eq
            [1],
262         nbr_of_particles, nbr_of_timesteps);
263
264     printf("Temp: %f\nHeat capacity: %.10f \t %.10f\n", temperature_eq[1],
265         heat_capacity_kin, heat_capacity_pot);
266
267     // Save results to file
268     file = fopen("heat_capacity.dat", "w");
269     fprintf(file, "%.2f\t%e\t%e\n", temperature_eq[1],
270         heat_capacity_kin, heat_capacity_pot);
271     fclose(file);
272
273
```

```
274
275     free(energy_kin);       energy_kin = NULL;
276     free(energy_pot);       energy_pot = NULL;
277     free(disp_arr);         disp_arr = NULL;
278     free(virial);           virial = NULL;
279     free(temperature_avg);  temperature_avg = NULL;
280     free(pressure_avg);     pressure_avg = NULL;
281     //free(temperature);        temperature = NULL;
282     //free(pressure);           pressure = NULL;
283
284     return 0;
285 }
```

A.5 Task6/MD_main.c

```
1  /*
2   MD_main.c
3
4   Created by Anders Lindman on 2013-10-31.
5   */
6
7  #include <stdio.h>
8  #include <math.h>
9  #include <stdlib.h>
10 #include <time.h>
11 #include "initfcc.h"
12 #include "alpotential.h"
13 #define nbr_of_particles 256
14 #define nbr_of_timesteps 1e4
15 #define nbr_of_timesteps_eq 4000
16 #define nbr_of_dimensions 3
17
18 double boundary_condition(double,double);
19
20
21
22 /* Main program */
23 int main()
24 {
25     srand(time(NULL));
26
27     /* Simulation parameters */
28     double m_AL; // Mass of atom
29     double cell_length; // Side length of supercell
30     double volume;
31     double lattice_spacing; // Smallest length between atoms
32     double initial_displacement;    // Initial displacement of the atoms from ↩
           their
33                                     // lattice positions
34     double lattice_param;   // Lattice parameter, length of each side in the
35                             // unit cell
36     double timestep;
37     double temperature_eq[] = { 1000.0+273.15, 700.0+273.15 };
38     double delta_temperature[] = { -10.0, 10.0 };
39     double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
40
41     FILE *file;
42
43
44     /* Current displacement, velocities, and acceleratons */
45     double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
46     double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
47     double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces
48
49     double heat_capacity;
50     double energy_avg[2] = { 0 };
51     double temperature_avg[2] = { 0 };
52
53
54     /* Allocate memory for large vectors */
55
56     double* energy_pot      = (double*) malloc(nbr_of_timesteps * sizeof(double)↩
           );
57     double* energy_kin      = (double*) malloc(nbr_of_timesteps * sizeof(double)↩
           );
58
59
60     /* Initialize parameters*/
61     initial_displacement    = 0.05;
62     lattice_param           = 4.046; // For aluminium (  )
63     lattice_spacing         = lattice_param/sqrt(2.0);
64     timestep                = 0.001; // 0.1 Bad, 0.01 Seems decent
65     m_AL                    = 0.0027964; // In ASU
66     cell_length             = 4*lattice_param;  // Side of the supercell: The ↩
           256 atoms are
```

```
                                                // structured in a block of 4x4 x4 unit cells
    volume                  = pow(cell_length, 3);



    /* Put atoms on lattice */
    init_fcc(q, 4, lattice_param);


    /* Initial conditions */
    for (int i = 0; i < nbr_of_particles; i++){
        for (int j = 0; j < nbr_of_dimensions; j++){

            // Initial perturbation from equilibrium
            q[i][j] += lattice_spacing * initial_displacement
                * ((double)rand()/(double)RAND_MAX);

        }
    }


    get_forces_AL(f, q, cell_length, nbr_of_particles);

    /* Simulation */
    /* Equilibrium stage */

    double inst_temperature_eq;
    double inst_pressure_eq;
    double alpha_T = 1.0;
    double alpha_P = 1.0;
    double energy_kin_eq = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,m_AL);
    double virial_eq = get_virial_AL(q,cell_length,nbr_of_particles);

    for (int d = 0; d < 2; d++) {

        for (int equil = 0; equil < 2; equil++) {

            double target_temp = temperature_eq[equil] + delta_temperature[d];

            for (int i = 1; i < nbr_of_timesteps_eq; i++)
            {
                /** Verlet algorithm **/
                /* Half step for velocity */
                for (int j = 0; j < nbr_of_particles; j++){
                    for (int k = 0; k < nbr_of_dimensions; k++){
                        v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
                    }
                }

                /* Update displacement*/
                for (int j = 0; j < nbr_of_particles; j++){
                    for (int k = 0; k < nbr_of_dimensions; k++){
                        q[j][k] += timestep * v[j][k];
                    }
                }

                /* Forces */
                get_forces_AL(f,q,cell_length,nbr_of_particles);

                /* Final velocity*/
                for (int j = 0; j < nbr_of_particles; j++){
                    for (int k = 0; k < nbr_of_dimensions; k++){
                        v[j][k] += timestep * 0.5* f[j][k]/m_AL;
                    }
                }

                /* Calculate energy */
                // Kinetic energy
                energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL);

                virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);


                inst_temperature_eq = instantaneous_temperature(energy_kin_eq, nbr_of_particles);
                inst_pressure_eq = instantaneous_pressure(virial_eq, inst_temperature_eq,
                                   nbr_of_particles, volume);


                // Update alhpas
                alpha_T = 1.0 + 0.01*(target_temp-inst_temperature_eq)/inst_temperature_eq;
                alpha_P = 1.0 - 0.01*(pressure_eq - inst_pressure_eq);

                // Scale velocities
                for (int j = 0; j < nbr_of_particles; j++){
                    for (int k = 0; k < nbr_of_dimensions; k++){
```

```
                        v[j][k] *= sqrt(alpha_T);
                    }
                }

                // Scale positions and volume
                cell_length *= pow(alpha_P, 1.0/3.0);
                volume = pow(cell_length, 3);
                for (int j = 0; j < nbr_of_particles; j++) {
                    for (int k = 0; k < nbr_of_dimensions; k++) {
                        q[j][k] *= pow(alpha_P, 1.0/3.0);
                    }
                }

            }
        }


        // Compute energies, temperature etc. at equilibrium
        energy_pot[0] = get_energy_AL(q, cell_length, nbr_of_particles);
        energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, ↩
            m_AL);


        /* Simulation after equilibrium*/
        for (int i = 1; i < nbr_of_timesteps; i++)
        {
            /** Verlet algorithm **/
            /* Half step for velocity */
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
                }
            }

            /* Update displacement*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    q[j][k] += timestep * v[j][k];
                }
            }

            /* Update Forces */
            get_forces_AL(f, q, cell_length, nbr_of_particles);

            /* Final velocity*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
                }
            }

            /* Calculate energy */
            // Potential energy
            energy_pot[i] = get_energy_AL(q, cell_length, nbr_of_particles);
            // Kinetic energy
            energy_kin[i] = get_kinetic_AL(v, nbr_of_dimensions, ↩
                nbr_of_particles, m_AL);


        } // equilibration/simulation

        // Compute heat capacity
        temperature_avg[d] = averaged_temperature(energy_kin, nbr_of_particles, ↩
            nbr_of_timesteps-1);
        // Compute average total energy
        for (int i = 0; i < nbr_of_timesteps; i++)
            energy_avg[d] += energy_pot[i] + energy_kin[i];
        energy_avg[d] /= nbr_of_timesteps;

        printf("Temp: %f\nAverage total energy: %.10f\n", temperature_avg[d], ↩
            energy_avg[d]);

    }

    // Compute heat capacity
    heat_capacity = (energy_avg[1]-energy_avg[0])/(temperature_avg[1]-↩
        temperature_avg[0]);

    printf("heat capacity: %f\n", heat_capacity);

    // Save results to file
    file = fopen("heat_capacity.dat", "w");
    fprintf(file, "%.2f\t%e\n", temperature_eq[1], heat_capacity);
    fclose(file);



    free(energy_kin);       energy_kin = NULL;
    free(energy_pot);       energy_pot = NULL;

    return 0;
```

```
}
```

A.6 Task7/MD_main.c

```
/*
 MD_main.c

 Created by Anders Lindman on 2013-10-31.
 */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "initfcc.h"
#include "alpotential.h"
#define nbr_of_particles 256
#define nbr_of_timesteps 1000
#define nbr_of_timesteps_eq 4000
#define nbr_of_dimensions 3

#define PI 3.141592653589
int get_bin(double , double , double , double );

double boundary_condition_dist_sq(double u1[3], double u2[3], double L);

/* Main program */
int main()
{
    srand(time(NULL));

    /* Simulation parameters */
    double m_AL; // Mass of atom
    double cell_length; // Side length of supercell
    double volume;
    double lattice_spacing; // Smallest length between atoms
    double initial_displacement;    // Initial displacement of the atoms from their
                                    // lattice positions
    double lattice_param;   // Lattice parameter, length of each side in the
                            // unit cell
    double timestep;
    double temperature_eq[] = { 1500.0+273.15, 700.0+273.15 };
    double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
    double isothermal_compressibility = 1.0; //0.8645443196; // 1.385e-11 m^2/N = 1.385/1.602    ^3/eV

    FILE *file;


    /* Current displacement, velocities, and acceleratons */
    double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
    double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
    double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces

    /* Allocate memory for large vectors */
    /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional array*/
    #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+nbr_of_dimensions*j+k])
    double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*nbr_of_dimensions*sizeof(double));

    double* energy          = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* energy_kin      = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* virial          = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* pressure_avg    = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* temperature     = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));
    double* pressure        = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));


    int k_bins  = 250;

    //TODO go over parameters again
    /* Initialize parameters*/
    initial_displacement    = 0.05;
    lattice_param           = 4.046; // For aluminium (  )
    lattice_spacing         = lattice_param/sqrt(2.0);
```

```
    timestep                = 0.01; // 0.1 Bad, 0.01 Seems decent
    m_AL                    = 0.0027964; // In ASU
    cell_length             = 4*lattice_param;  // Side of the supercell: The 256 atoms are
                                                // structured in a block of 4x4x4 unit cells
    volume                  = pow(cell_length, 3);

    // Initialize all displacements, for all times, as 0
    for (int i  = 0; i < nbr_of_timesteps; i++) {
        for (int j = 0; j < nbr_of_particles; j++) {
            for (int k = 0; k < nbr_of_dimensions; k++) {
                qq(i,j,k) = 0;
            }
        }
    }

    /* Put atoms on lattice */
    init_fcc(q, 4, lattice_param);


    /* Initial conditions */
    for (int i = 0; i < nbr_of_particles; i++) {
        for (int j = 0; j < nbr_of_dimensions; j++) {

            // Initial perturbation from equilibrium
            q[i][j] += lattice_spacing * initial_displacement
                * ((double)rand()/(double)RAND_MAX);

        }
    }


    get_forces_AL(f, q, cell_length, nbr_of_particles);

    /* Simulation */
    /* Equilibrium stage */

    double inst_temperature_eq;
    double inst_pressure_eq;
    double alpha_T = 1.0;
    double alpha_P = 1.0;
    double energy_kin_eq = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,m_AL);
    double virial_eq = get_virial_AL(q,cell_length,nbr_of_particles);

    temperature[0]  = instantaneous_temperature(energy_kin_eq, nbr_of_particles);
    pressure[0]     = instantaneous_pressure(virial_eq, temperature[0], nbr_of_particles, volume);

    for (int equil = 0; equil < 2; equil++) {
        for (int i = 1; i < nbr_of_timesteps_eq; i++) {

            /** Verlet algorithm **/
            /* Half step for velocity */
            for (int j = 0; j < nbr_of_particles; j++) {
                for (int k = 0; k < nbr_of_dimensions; k++) {
                    v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
                }
            }

            /* Update displacement*/
            for (int j = 0; j < nbr_of_particles; j++) {
                for (int k = 0; k < nbr_of_dimensions; k++) {
                    q[j][k] += timestep * v[j][k];
                }
            }

            /* Forces */
            get_forces_AL(f,q,cell_length,nbr_of_particles);

            /* Final velocity*/
            for (int j = 0; j < nbr_of_particles; j++) {
                for (int k = 0; k < nbr_of_dimensions; k++) {
                    v[j][k] += timestep * 0.5* f[j][k]/m_AL;
                }
            }

            /* Calculate energy */
            // Kinetic energy
            energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL);

            virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);


            inst_temperature_eq = instantaneous_temperature(energy_kin_eq, nbr_of_particles);
            temperature[equil*(nbr_of_timesteps_eq-1) + i] = inst_temperature_eq;
```

```
            inst_pressure_eq = instantaneous_pressure(virial_eq, ↩
                 inst_temperature_eq,
                nbr_of_particles, volume);
            pressure[equil*(nbr_of_timesteps_eq-1) + i] = inst_pressure_eq;


            // Update alhpas
            alpha_T = 1.0 + 0.01*(temperature_eq[equil]-inst_temperature_eq)/↩
                 inst_temperature_eq;
            alpha_P = 1.0 - 0.01*isothermal_compressibility*(pressure_eq - ↩
                 inst_pressure_eq);


            // Scale velocities
            for (int j = 0; j < nbr_of_particles; j++) {
                for (int k = 0; k < nbr_of_dimensions; k++) {
                    v[j][k] *= sqrt(alpha_T);
                }
            }

            // Scale positions and volume
            cell_length *= pow(alpha_P, 1.0/3.0);
            volume = pow(cell_length, 3);
            for (int j = 0; j < nbr_of_particles; j++) {
                for (int k = 0; k < nbr_of_dimensions; k++) {
                    q[j][k] *= pow(alpha_P, 1.0/3.0);
                }
            }

        }
    }

    for (int i = 0; i < nbr_of_particles; i++) {
        for (int j = 0; j < nbr_of_dimensions; j++) {
            qq(0,i,j)=q[i][j];
        }
    }


    // Compute energies, temperature etc. at equilibrium
    double min = 0.0;
    double max = sqrt(3*cell_length*cell_length);
    double d_r = (max-min)/(1.0*k_bins);
    int bins[k_bins];
    int* bins2 = (int*) malloc(k_bins * sizeof(int));

    for (int i = 0; i < k_bins; i++) {
        bins[i]=0;
        bins2[i]=0;
    }

    for (int i = 1; i < nbr_of_timesteps; i++)
    {
        /** Verlet algorithm **/
        /* Half step for velocity */
        for (int j = 0; j < nbr_of_particles; j++){
            for (int k = 0; k < nbr_of_dimensions; k++){
                v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
            }
        }

        /* Update displacement*/
        for (int j = 0; j < nbr_of_particles; j++){
            for (int k = 0; k < nbr_of_dimensions; k++){
                q[j][k] += timestep * v[j][k];
            }
        }

        /* Forces */
        get_forces_AL(f,q,cell_length,nbr_of_particles);

        /* Final velocity*/
        for (int j = 0; j < nbr_of_particles; j++){
            for (int k = 0; k < nbr_of_dimensions; k++){
                v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
            }
        }

        /* Calculate energy */
        // Potential energy
        energy[i] = get_energy_AL(q,cell_length,nbr_of_particles);
        // Kinetic energy
        energy_kin[i] = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,m_AL↩
             );

        virial[i]=get_virial_AL(q,cell_length,nbr_of_particles);

        /* Save current displacements to array*/
        for (int j = 0; j < nbr_of_particles; j++){
            for (int k = 0; k < nbr_of_dimensions; k++){
                qq(i,j,k)=q[j][k];
```

```
            }
        }
    }


    // Create Histogram

    for (int i = 1; i < nbr_of_timesteps; i++)
    {
        for (int j = 1 ; j < nbr_of_particles; j++) {
            for (int k = j+1 ; k < nbr_of_particles; k++) {

                double q1[nbr_of_dimensions];
                double q2[nbr_of_dimensions];
                for (int d = 0; d < nbr_of_dimensions; d++) {
                    q1[d] = qq(i,j,d);
                    q2[d] = qq(i,k,d);
                }
                double distance_sq = boundary_condition_dist_sq(q1, q2, ↩
                     cell_length);
                double dist = sqrt(distance_sq);
                int bin = get_bin(dist,min,max,d_r);
                bins2[bin] += 2;
            }
        }
    }
    double Nideal[k_bins];
    double factor =((double)(nbr_of_particles-1.0))/volume * 4.0*PI/3.0;
    for (int i = 0; i < k_bins; i++) {
        Nideal[i] = factor*(3.0*i*i-3.0*i+1.0)*d_r*d_r*d_r;
    }




    /* Save data to file*/
    file = fopen("histogram.dat","w");
    for (int i = 0; i < k_bins; i ++) {
        fprintf(file, "%e \t %i \t %i \t %e \n",d_r*(i-0.5), bins[i],bins2[i], ↩
             Nideal[i]);
    }
    fclose(file);
    // TO THIS ISH TODO


    free(energy_kin);       energy_kin = NULL;
    free(energy);           energy = NULL;
    free(disp_arr);         disp_arr = NULL;
    free(virial);           virial = NULL;
    free(temperature_avg);  temperature_avg = NULL;
    free(pressure_avg);     pressure_avg = NULL;

    return 0;
}

int get_bin(double val , double min , double max , double  d_r)
{
    int bin = 0;
    double current = min;
    while (current <= val)
    {
        current += d_r;
        bin++;
    }
    if (current > max)
        return --bin;
    return bin;
}

double boundary_condition_dist_sq(double u1[3], double u2[3], double L)
{
    double d[3];
    for (int i = 0; i < 3; i++) {
        u1[i] /= L;
        u2[i] /= L;

        u1[i] -= floor(u1[i]);
        d[i] = u1[i] - (u2[i] - floor(u2[i]));
        d[i] -= (double)((int)floor(d[i]+0.5));
    }

    double sum = 0.0;
    for (int i = 0; i < 3; i++)
        sum += pow(d[i], 2);
    return L*L * sum;

}
```

A.7 Task8/MD_main.c

```
/*
 MD_main.c

 Created by Anders Lindman on 2013-10-31.
 */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "initfcc.h"
#include "alpotential.h"
#include <complex.h>
#define nbr_of_particles 256
#define nbr_of_timesteps 1000
#define nbr_of_timesteps_eq 4000
#define nbr_of_dimensions 3

#define PI 3.141592653589
int get_bin(double , double , double , double );

double boundary_condition(double,double);

/* Main program */
int main()
{
    srand(time(NULL));

    /* Simulation parameters */
    double m_AL; // Mass of atom
    double cell_length; // Side length of supercell
    double volume;
    double lattice_spacing; // Smallest length between atoms
    double initial_displacement;    // Initial displacement of the atoms from their
                                    // lattice positions
    double lattice_param;   // Lattice parameter, length of each side in the
                            // unit cell
    double timestep;
    double temperature_eq[] = { 1500.0+273.15, 700.0+273.15 };
    double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
    double isothermal_compressibility = 1.0; //0.8645443196; // 1.385e-11 m^2/N = 1.385/1.602    ^3/eV

    FILE *file;


    /* Current displacement, velocities, and acceleratons */
    double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
    double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
    double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces

    /* Allocate memory for large vectors */
    /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional array*/
    #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+nbr_of_dimensions*j+k])
    double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*nbr_of_dimensions*sizeof(double));

    double* energy          = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* energy_kin      = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* virial          = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* pressure_avg    = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* temperature     = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));
    double* pressure        = (double*) malloc((2 * nbr_of_timesteps_eq + nbr_of_timesteps) * sizeof(double));



    //TODO go over parameters again
    /* Initialize parameters*/
    initial_displacement    = 0.05;
    lattice_param           = 4.046; // For aluminium (  )
    lattice_spacing         = lattice_param/sqrt(2.0);
    timestep                = 0.01; // 0.1 Bad, 0.01 Seems decent
    m_AL                    = 0.0027964; // In ASU
    cell_length             = 4*lattice_param;  // Side of the supercell: The 256 atoms are
```

```
                                            // structured in a block of 4x4 unit cells
    volume                  = pow(cell_length, 3);

    // Initialize all displacements, for all times, as 0
    for (int i  = 0; i < nbr_of_timesteps; i++){
        for (int j = 0; j < nbr_of_particles; j++){
            for (int k = 0; k < nbr_of_dimensions; k++){
                qq(i,j,k) = 0;
            }
        }
    }

    /* Put atoms on lattice */
    init_fcc(q, 4, lattice_param);


    /* Initial conditions */
    for (int i = 0; i < nbr_of_particles; i++){
        for (int j = 0; j < nbr_of_dimensions; j++){

            // Initial perturbation from equilibrium
            q[i][j] += lattice_spacing * initial_displacement
                * ((double)rand()/(double)RAND_MAX);

        }
    }


    get_forces_AL(f, q, cell_length, nbr_of_particles);

    /* Simulation */
    /* Equilibrium stage */

    double inst_temperature_eq;
    double inst_pressure_eq;
    double alpha_T = 1.0;
    double alpha_P = 1.0;
    double energy_kin_eq = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,
        m_AL);
    double virial_eq = get_virial_AL(q,cell_length,nbr_of_particles);

    temperature[0]  = instantaneous_temperature(energy_kin_eq, nbr_of_particles)
        ;
    pressure[0]     = instantaneous_pressure(virial_eq, temperature[0],
        nbr_of_particles, volume);

    for (int equil = 0; equil < 2; equil++) {
        for (int i = 1; i < nbr_of_timesteps_eq; i++)
        {
            /** Verlet algorithm **/
            /* Half step for velocity */
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
                }
            }

            /* Update displacement*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    q[j][k] += timestep * v[j][k];
                }
            }

            /* Forces */
            get_forces_AL(f,q,cell_length,nbr_of_particles);

            /* Final velocity*/
            for (int j = 0; j < nbr_of_particles; j++){
                for (int k = 0; k < nbr_of_dimensions; k++){
                    v[j][k] += timestep * 0.5* f[j][k]/m_AL;
                }
            }

            /* Calculate energy */
            // Kinetic energy
            energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions,
                nbr_of_particles, m_AL);

            virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);


            inst_temperature_eq = instantaneous_temperature(energy_kin_eq,
                nbr_of_particles);
            temperature[equil*(nbr_of_timesteps_eq-1) + i] = inst_temperature_eq
                ;
            inst_pressure_eq = instantaneous_pressure(virial_eq,
                inst_temperature_eq,
                nbr_of_particles, volume);
            pressure[equil*(nbr_of_timesteps_eq-1) + i] = inst_pressure_eq;
```

```
157
158
159             // Update alhpas
160             alpha_T = 1.0 + 0.01*(temperature_eq[equil]-inst_temperature_eq)/↩
                    inst_temperature_eq;
161             alpha_P = 1.0 - 0.01*isothermal_compressibility*(pressure_eq - ↩
                    inst_pressure_eq);
162
163             // DEBUG:alpha
164             //printf("%.8f \t %.8f \n", alpha_T, alpha_P);
165
166             // Scale velocities
167             for (int j = 0; j < nbr_of_particles; j++){
168                 for (int k = 0; k < nbr_of_dimensions; k++){
169                     v[j][k] *= sqrt(alpha_T);
170                 }
171             }
172
173             // Scale positions and volume
174             cell_length *= pow(alpha_P, 1.0/3.0);
175             volume = pow(cell_length, 3);
176             for (int j = 0; j < nbr_of_particles; j++) {
177                 for (int k = 0; k < nbr_of_dimensions; k++) {
178                     q[j][k] *= pow(alpha_P, 1.0/3.0);
179                 }
180             }
181
182         }
183     }
184
185     for (int i = 0; i < nbr_of_particles; i++){
186         for (int j = 0; j < nbr_of_dimensions; j++){
187             qq(0,i,j)=q[i][j];
188         }
189     }
190
191     // Compute energies, temperature etc. at equilibrium
192     energy[0] = get_energy_AL(q, cell_length, nbr_of_particles);
193     virial[0] = get_virial_AL(q, cell_length, nbr_of_particles);
194     energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL)↩
            ;
195     temperature_avg[0] = instantaneous_temperature(energy_kin[0], ↩
            nbr_of_particles);
196     pressure_avg[0] = instantaneous_pressure(virial[0], temperature_avg[0],
197         nbr_of_particles, volume);
198
199     /* Simulation after equilibrium*/
200     for (int i = 1; i < nbr_of_timesteps; i++)
201     {
202         /** Verlet algorithm **/
203         /* Half step for velocity */
204         for (int j = 0; j < nbr_of_particles; j++){
205             for (int k = 0; k < nbr_of_dimensions; k++){
206                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
207             }
208         }
209
210         /* Update displacement*/
211         for (int j = 0; j < nbr_of_particles; j++){
212             for (int k = 0; k < nbr_of_dimensions; k++){
213                 q[j][k] += timestep * v[j][k];
214             }
215         }
216
217         /* Update Forces */
218         get_forces_AL(f, q, cell_length, nbr_of_particles);
219
220         /* Final velocity*/
221         for (int j = 0; j < nbr_of_particles; j++){
222             for (int k = 0; k < nbr_of_dimensions; k++){
223                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
224             }
225         }
226
227         /* Calculate energy */
228         // Potential energy
229         energy[i] = get_energy_AL(q, cell_length, nbr_of_particles);
230         // Kinetic energy
231         energy_kin[i] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, ↩
                m_AL);
232
233         virial[i] = get_virial_AL(q, cell_length, nbr_of_particles);
234
235         // Temperature
236         temperature_avg[i] = averaged_temperature(energy_kin, nbr_of_particles, ↩
                i);
237         temperature[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_temperature(↩
                energy_kin[i],
238             nbr_of_particles);
239
240
```

```
        // Pressure
        pressure_avg[i] = averaged_pressure(virial, energy_kin, volume, i);
        pressure[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_pressure(virial[↩
             i],
            temperature[2*(nbr_of_timesteps_eq-1) + i],
            nbr_of_particles, volume);


        /* Save current displacements to array*/
        for (int j = 0; j < nbr_of_particles; j++){
            for (int k = 0; k < nbr_of_dimensions; k++){
                qq(i,j,k)=q[j][k];
            }
        }
    } // equilibration/simulation

    int n_x = 30;
    int n_y = 30;
    int n_z = 30;

    double factor = PI*2.0/cell_length;

    double qS[n_x][n_y][n_z][3];
    for (int i = 0; i < n_x; i++)
        for (int j = 0; j < n_y ; j++)
            for (int k = 0; k < n_z; k++){
                    qS[i][j][k][0]=i*factor;
                    qS[i][j][k][1]=j*factor;
                    qS[i][j][k][2]=k*factor;
                }

    double s[n_x][n_y][n_z];
    for (int i = 0; i < n_x; i++)
        for (int j = 0; j < n_y ; j++)
            for (int k = 0; k < n_z; k++)
            {
                if ( !((i==j) && (i==k) && (i==0))){
                    double complex sum = 0;
                    for (int r=0; r < nbr_of_particles; r++)
                    {
                        double complex expo=0;
                        for (int d = 0; d < nbr_of_dimensions; d++)
                        {
                            double ri = q[r][d];
                            ri=boundary_condition(ri,cell_length);
                            expo+= qS[i][j][k][d]*ri;
                        }
                        expo=expo*I;
                        sum+= cexp(expo);
                    }
                    sum = cabs(sum);
                    sum=sum*sum/nbr_of_particles;
                    s[i][j][k]=sum;
                }
            }

    double data[n_x*n_y*n_z];
    double dis[n_x*n_y*n_z];
    int iterator =0;
    for (int i = 0; i < n_x; i++)
        for (int j = 0; j < n_y ; j++)
            for (int k = 0; k < n_z; k++)
            {
                dis[iterator] =sqrt(1.0*i*i+1.0*j*j+1.0*k*k);
                data[iterator] = s[i][j][k];
                iterator++;
            }
    double max =0;
    double min = 1e10;
    for (int i = 0; i < n_x*n_y*n_z; i++ )
    {
        if (dis[i] > max)
            max = dis[i];
        if (dis[i] < min)
            min = dis[i];
    }

    int k_bins=200;
    double d_r = (max-min)/(1.0*k_bins);
    int bins[k_bins];
    for (int i = 0; i < n_x*n_y*n_z; i++)
    {
        int bin = get_bin(data[i],min,max,d_r);
        bins[bin]++;
    }

    file = fopen("data.dat","w");
    for (int i = 0; i < k_bins; i++)
    {
        fprintf(file, "%e \t %i \n", (double)(min+d_r*i*1.0), bins[i]);
    }
```

```
    fclose(file);
    /*
    file = fopen("data.dat","w");
    for (int i = 0; i < n_x*n_y*n_z; i ++)
    {
        fprintf(file, "%e \t %e \n",dis[i],data[i] );
    }
    fclose(file);*/

    free(energy_kin);       energy_kin = NULL;
    free(energy);           energy = NULL;
    free(disp_arr);         disp_arr = NULL;
    free(virial);           virial = NULL;
    free(temperature_avg);  temperature_avg = NULL;
    free(pressure_avg);     pressure_avg = NULL;

    return 0;
}

int get_bin(double val , double min , double max , double  d_r)
{
    int bin =0;
    double current=min;
    while (current <= val)
    {
        current += d_r;
        bin++;
    }
    return bin;
}

double boundary_condition(double u, double L)
{

    double f = fmod(u,L);
    if (f < 0)
        return -f;
    else
        return f;
}
```

```
/*
 alpotential.c
 Program that contains functions that calculate properties (potential energy, forces, etc.) of a set of Aluminum atoms using an embedded atom model (EAM) potential.
 Created by Anders Lindman on 2013-03-14.
 */


#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/*Parameters for the AL EAM potential */
#define PAIR_POTENTIAL_ROWS 18
const double pair_potential[90] = {2.0210, 2.2730, 2.4953, 2.7177, 2.9400,
     3.1623, 3.3847, 3.6070, 3.8293, 4.0517, 4.2740, 4.4963, 4.7187, 4.9410,
     5.1633, 5.3857, 5.6080, 6.0630, 2.0051, 0.7093, 0.2127, 0.0202, -0.0386,
     -0.0492, -0.0424, -0.0367, -0.0399, -0.0574, -0.0687, -0.0624, -0.0492,
     -0.0311, -0.0153, -0.0024, -0.0002, 0, -7.2241, -3.3383, -1.3713, -0.4753,
     -0.1171, 0.0069, 0.0374, 0.0122, -0.0524, -0.0818, -0.0090, 0.0499, 0.0735,
      0.0788, 0.0686, 0.0339, -0.0012, 0, 9.3666, 6.0533, 2.7940, 1.2357,
     0.3757, 0.1818, -0.0445, -0.0690, -0.2217, 0.0895, 0.2381, 0.0266, 0.0797,
     -0.0557, 0.0097, -0.1660, 0.0083, 0, -4.3827, -4.8865, -2.3363, -1.2893,
     -0.2907, -0.3393, -0.0367, -0.2290, 0.4667, 0.2227, -0.3170, 0.0796,
     -0.2031, 0.0980, -0.2634, 0.2612, -0.0102, 0};


#define ELECTRON_DENSITY_ROWS 15
const double electron_density[75] = {2.0210, 2.2730, 2.5055, 2.7380, 2.9705,
     3.2030, 3.4355, 3.6680, 3.9005, 4.1330, 4.3655, 4.5980, 4.8305, 5.0630,
     6.0630, 0.0824, 0.0918, 0.0883, 0.0775, 0.0647, 0.0512, 0.0392, 0.0291,
     0.0186, 0.0082, 0.0044, 0.0034, 0.0027, 0.0025, 0.0000, 0.0707, 0.0071,
     -0.0344, -0.0533, -0.0578, -0.0560, -0.0465, -0.0428, -0.0486, -0.0318,
     -0.0069, -0.0035, -0.0016, -0.0008, 0, -0.1471, -0.1053, -0.0732, -0.0081,
     -0.0112, 0.0189, 0.0217, -0.0056, -0.0194, 0.0917, 0.0157, -0.0012, 0.0093,
      -0.0059, 0, 0.0554, 0.0460, 0.0932, -0.0044, 0.0432, 0.0040, -0.0392,
     -0.0198, 0.1593, -0.1089, -0.0242, 0.0150, -0.0218, 0.0042, 0};

#define EMBEDDING_ENERGY_ROWS 13
const double embedding_energy[65] = {0, 0.1000, 0.2000, 0.3000, 0.4000, 0.5000,
     0.6000, 0.7000, 0.8000, 0.9000, 1.0000, 1.1000, 1.2000, 0, -1.1199,
     -1.4075, -1.7100, -1.9871, -2.2318, -2.4038, -2.5538, -2.6224, -2.6570,
     -2.6696, -2.6589, -2.6358, -18.4387, -5.3706, -2.3045, -3.1161, -2.6175,
     -2.0666, -1.6167, -1.1280, -0.4304, -0.2464, -0.0001, 0.1898, 0.2557,
     86.5178, 44.1632, -13.5018, 5.3853, -0.3996, 5.9090, -1.4103, 6.2976,
```

```
     0.6785, 1.1611, 1.3022, 0.5971, 0.0612, -141.1819, -192.2166, 62.9570, ↩
     -19.2831, 21.0288, -24.3978, 25.6930, -18.7304, 1.6087, 0.4704, -2.3503, ↩
     -1.7862, -1.7862};


#define k_b 0.00008617 // (eV)

/* Evaluates the spline in x. */

double splineEval(double x, const double *table,int m) {
    /* int m = mxGetM(spline), i, k;*/
    int i, k;

  /*double *table = mxGetPr(spline);*/
    double result;

    int k_lo = 0, k_hi = m;

    /* Find the index by bisection. */
    while (k_hi - k_lo > 1) {
        k = (k_hi + k_lo) >> 1;
        if (table[k] > x)
            k_hi = k;
        else
            k_lo = k;
    }

    /* Switch to local coord. */
    x -= table[k_lo];

    /* Horner's scheme */
    result = table[k_lo + 4*m];
    for (i = 3; i > 0; i--) {
        result *= x;
        result += table[k_lo + i*m];
    }

    return result;
}

/* Evaluates the derivative of the spline in x. */

double splineEvalDiff(double x, const double *table, int m) {
    /*int m = mxGetM(spline), i, k;
   double *table = mxGetPr(spline);
   */
  int i, k;
  double result;

    int k_lo = 0, k_hi = m;

    /* Find the index by bisection. */
    while (k_hi - k_lo > 1) {
        k = (k_hi + k_lo) >> 1;
        if (table[k] > x)
            k_hi = k;
        else
            k_lo = k;
    }

    /* Switch to local coord. */
    x -= table[k_lo];

    /* Horner's scheme */
    result = 3*table[k_lo + 4*m];
    for (i = 3; i > 1; i--) {
        result *= x;
        result += (i-1)*table[k_lo + i*m];
    }

    return result;
}

/* Returns the forces */
void get_forces_AL(double forces[][3], double positions[][3], double cell_length↩
     , int nbr_atoms)
{
  int i, j;
  double cell_length_inv, cell_length_sq;
  double rcut, rcut_sq;
  double densityi, dens, drho_dr, force;
  double dUpair_dr;
  double sxi, syi, szi, sxij, syij, szij, rij,  rij_sq;

  double *sx = malloc(nbr_atoms * sizeof (double));
  double *sy = malloc(nbr_atoms * sizeof (double));
  double *sz = malloc(nbr_atoms * sizeof (double));
  double *fx = malloc(nbr_atoms * sizeof (double));
  double *fy = malloc(nbr_atoms * sizeof (double));
  double *fz = malloc(nbr_atoms * sizeof (double));

```

```
109   double *density = malloc(nbr_atoms * sizeof (double));
110   double *dUembed_drho = malloc(nbr_atoms * sizeof (double));
111
112   rcut = 6.06;
113   rcut_sq = rcut * rcut;
114
115   cell_length_inv = 1 / cell_length;
116   cell_length_sq = cell_length * cell_length;
117
118   for (i = 0; i < nbr_atoms; i++){
119     sx[i] = positions[i][0] * cell_length_inv;
120     sy[i] = positions[i][1] * cell_length_inv;
121     sz[i] = positions[i][2] * cell_length_inv;
122   }
123
124   for (i = 0; i < nbr_atoms; i++){
125     density[i] = 0;
126     fx[i] = 0;
127     fy[i] = 0;
128     fz[i] = 0;
129   }
130
131   for (i = 0; i < nbr_atoms; i++) {
132     /* Periodically translate coords of current particle to positive quadrants ↩
          */
133         sxi = sx[i] - floor(sx[i]);
134         syi = sy[i] - floor(sy[i]);
135         szi = sz[i] - floor(sz[i]);
136
137     densityi = density[i];
138
139         /* Loop over other atoms. */
140         for (j = i + 1; j < nbr_atoms; j++) {
141       /* Periodically translate atom j to positive quadrants and calculate ↩
            distance to it. */
142           sxij = sxi - (sx[j] - floor(sx[j]));
143           syij = syi - (sy[j] - floor(sy[j]));
144           szij = szi - (sz[j] - floor(sz[j]));
145
146       /* Periodic boundary conditions. */
147           sxij = sxij - (int)floor(sxij + 0.5);
148           syij = syij - (int)floor(syij + 0.5);
149           szij = szij - (int)floor(szij + 0.5);
150
151       /* squared distance between atom i and j */
152           rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
153
154       /* Add force and energy contribution if distance between atoms smaller ↩
            than rcut */
155           if (rij_sq < rcut_sq) {
156         rij = sqrt(rij_sq);
157         dens = splineEval(rij, electron_density, ELECTRON_DENSITY_ROWS);
158         densityi += dens;
159         density[j] += dens;
160       }
161     }
162     density[i] = densityi;
163   }
164
165   /* Loop over atoms to calculate derivative of embedding function
166    and embedding function. */
167     for (i = 0; i < nbr_atoms; i++) {
168         dUembed_drho[i] = splineEvalDiff(density[i], embedding_energy, ↩
              EMBEDDING_ENERGY_ROWS);
169     }
170
171   /* Compute forces on atoms. */
172     /* Loop over atoms again :-(. */
173
174   for (i = 0; i < nbr_atoms; i++) {
175     /* Periodically translate coords of current particle to positive quadrants ↩
          */
176         sxi = sx[i] - floor(sx[i]);
177         syi = sy[i] - floor(sy[i]);
178         szi = sz[i] - floor(sz[i]);
179
180     densityi = density[i];
181
182         /* Loop over other atoms. */
183         for (j = i + 1; j < nbr_atoms; j++) {
184       /* Periodically translate atom j to positive quadrants and calculate ↩
            distance to it. */
185           sxij = sxi - (sx[j] - floor(sx[j]));
186           syij = syi - (sy[j] - floor(sy[j]));
187           szij = szi - (sz[j] - floor(sz[j]));
188
189       /* Periodic boundary conditions. */
190           sxij = sxij - (int)floor(sxij + 0.5);
191           syij = syij - (int)floor(syij + 0.5);
192           szij = szij - (int)floor(szij + 0.5);
193
```

```
194       /* squared distance between atom i and j */
195             rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
196
197       /* Add force and energy contribution if distance between atoms smaller ↩
              than rcut */
198             if (rij_sq < rcut_sq) {
199         rij = sqrt(rij_sq);
200         dUpair_dr = splineEvalDiff(rij, pair_potential, PAIR_POTENTIAL_ROWS);
201         drho_dr = splineEvalDiff(rij, electron_density, ELECTRON_DENSITY_ROWS);
202
203         /* Add force contribution from i-j interaction */
204                 force = -(dUpair_dr + (dUembed_drho[i] + dUembed_drho[j])*↩
                        drho_dr) / rij;
205                 fx[i] += force * sxij * cell_length;
206                 fy[i] += force * syij * cell_length;
207                 fz[i] += force * szij * cell_length;
208                 fx[j] -= force * sxij * cell_length;
209                 fy[j] -= force * syij * cell_length;
210                 fz[j] -= force * szij * cell_length;
211             }
212     }
213   }
214
215   for (i = 0; i < nbr_atoms; i++){
216     forces[i][0] = fx[i];
217     forces[i][1] = fy[i];
218     forces[i][2] = fz[i];
219   }
220
221   free(sx); free(sy); free(sz); sx = NULL; sy = NULL; sz = NULL;
222   free(fx); free(fy); free(fz); fx = NULL; fy = NULL; fz = NULL;
223   free(density); density = NULL;
224   free(dUembed_drho); dUembed_drho = NULL;
225
226 }
227
228 /* Returns the potential energy */
229 double get_energy_AL(double positions[][3], double cell_length, int nbr_atoms)
230 {
231   int i, j;
232   double cell_length_inv, cell_length_sq;
233   double rcut, rcut_sq;
234   double energy;
235   double densityi, dens;
236   double sxi, syi, szi, sxij, syij, szij, rij,  rij_sq;
237
238   double *sx = malloc(nbr_atoms * sizeof (double));
239   double *sy = malloc(nbr_atoms * sizeof (double));
240   double *sz = malloc(nbr_atoms * sizeof (double));
241
242   double *density = malloc(nbr_atoms * sizeof (double));
243
244   rcut = 6.06;
245   rcut_sq = rcut * rcut;
246
247   cell_length_inv = 1 / cell_length;
248   cell_length_sq = cell_length * cell_length;
249
250   for (i = 0; i < nbr_atoms; i++){
251     sx[i] = positions[i][0] * cell_length_inv;
252     sy[i] = positions[i][1] * cell_length_inv;
253     sz[i] = positions[i][2] * cell_length_inv;
254   }
255
256   for (i = 0; i < nbr_atoms; i++){
257     density[i] = 0;
258   }
259
260   energy = 0;
261
262   for (i = 0; i < nbr_atoms; i++) {
263     /* Periodically translate coords of current particle to positive quadrants ↩
           */
264         sxi = sx[i] - floor(sx[i]);
265         syi = sy[i] - floor(sy[i]);
266         szi = sz[i] - floor(sz[i]);
267
268     densityi = density[i];
269
270         /* Loop over other atoms. */
271         for (j = i + 1; j < nbr_atoms; j++) {
272       /* Periodically translate atom j to positive quadrants and calculate ↩
              distance to it. */
273             sxij = sxi - (sx[j] - floor(sx[j]));
274             syij = syi - (sy[j] - floor(sy[j]));
275             szij = szi - (sz[j] - floor(sz[j]));
276
277       /* Periodic boundary conditions. */
278             sxij = sxij - (int)floor(sxij + 0.5);
279             syij = syij - (int)floor(syij + 0.5);
280             szij = szij - (int)floor(szij + 0.5);
```

```
281
282       /* squared distance between atom i and j */
283             rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
284
285       /* Add force and energy contribution if distance between atoms smaller ↩
              than rcut */
286             if (rij_sq < rcut_sq) {
287         rij = sqrt(rij_sq);
288         dens = splineEval(rij, electron_density, ELECTRON_DENSITY_ROWS);
289         densityi += dens;
290         density[j] += dens;
291
292         /* Add energy contribution from i-j interaction */
293         energy += splineEval(rij, pair_potential, PAIR_POTENTIAL_ROWS);
294
295       }
296     }
297     density[i] = densityi;
298   }
299
300   /* Loop over atoms to calculate derivative of embedding function
301    and embedding function. */
302     for (i = 0; i < nbr_atoms; i++) {
303         energy += splineEval(density[i], embedding_energy, EMBEDDING_ENERGY_ROWS↩
                ;
304     }
305
306   free(sx); free(sy); free(sz); sx = NULL; sy = NULL; sz = NULL;
307   free(density); density = NULL;
308
309   return(energy);
310
311 }
312
313 /* Returns the virial */
314 double get_virial_AL(double positions[][3], double cell_length, int nbr_atoms)
315 {
316   int i, j;
317   double cell_length_inv, cell_length_sq;
318   double rcut, rcut_sq;
319   double virial;
320   double densityi, dens, drho_dr, force;
321   double dUpair_dr;
322   double sxi, syi, szi, sxij, syij, szij, rij, rij_sq;
323
324   double *sx = malloc(nbr_atoms * sizeof (double));
325   double *sy = malloc(nbr_atoms * sizeof (double));
326   double *sz = malloc(nbr_atoms * sizeof (double));
327
328   double *density = malloc(nbr_atoms * sizeof (double));
329   double *dUembed_drho = malloc(nbr_atoms * sizeof (double));
330
331   rcut = 6.06;
332   rcut_sq = rcut * rcut;
333
334   cell_length_inv = 1 / cell_length;
335   cell_length_sq = cell_length * cell_length;
336
337   for (i = 0; i < nbr_atoms; i++){
338     sx[i] = positions[i][0] * cell_length_inv;
339     sy[i] = positions[i][1] * cell_length_inv;
340     sz[i] = positions[i][2] * cell_length_inv;
341   }
342
343   for (i = 0; i < nbr_atoms; i++){
344     density[i] = 0;
345   }
346
347   for (i = 0; i < nbr_atoms; i++) {
348     /* Periodically translate coords of current particle to positive quadrants ↩
           */
349         sxi = sx[i] - floor(sx[i]);
350         syi = sy[i] - floor(sy[i]);
351         szi = sz[i] - floor(sz[i]);
352
353     densityi = density[i];
354
355         /* Loop over other atoms. */
356         for (j = i + 1; j < nbr_atoms; j++) {
357       /* Periodically translate atom j to positive quadrants and calculate ↩
              distance to it. */
358             sxij = sxi - (sx[j] - floor(sx[j]));
359             syij = syi - (sy[j] - floor(sy[j]));
360             szij = szi - (sz[j] - floor(sz[j]));
361
362       /* Periodic boundary conditions. */
363             sxij = sxij - (int)floor(sxij + 0.5);
364             syij = syij - (int)floor(syij + 0.5);
365             szij = szij - (int)floor(szij + 0.5);
366
367       /* squared distance between atom i and j */
```

```
            rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);

      /* Add force and energy contribution if distance between atoms smaller than rcut */
            if (rij_sq < rcut_sq) {
        rij = sqrt(rij_sq);
        dens = splineEval(rij, electron_density, ELECTRON_DENSITY_ROWS);
        densityi += dens;
        density[j] += dens;
      }
    }
    density[i] = densityi;
  }

  /* Loop over atoms to calculate derivative of embedding function
   and embedding function. */
    for (i = 0; i < nbr_atoms; i++) {
        dUembed_drho[i] = splineEvalDiff(density[i], embedding_energy, EMBEDDING_ENERGY_ROWS);
    }

  /* Compute forces on atoms. */
    /* Loop over atoms again :-(. */

  virial = 0;

  for (i = 0; i < nbr_atoms; i++) {
    /* Periodically translate coords of current particle to positive quadrants */
        sxi = sx[i] - floor(sx[i]);
        syi = sy[i] - floor(sy[i]);
        szi = sz[i] - floor(sz[i]);

    densityi = density[i];

        /* Loop over other atoms. */
        for (j = i + 1; j < nbr_atoms; j++) {
      /* Periodically translate atom j to positive quadrants and calculate distance to it. */
            sxij = sxi - (sx[j] - floor(sx[j]));
            syij = syi - (sy[j] - floor(sy[j]));
            szij = szi - (sz[j] - floor(sz[j]));

      /* Periodic boundary conditions. */
            sxij = sxij - (int)floor(sxij + 0.5);
            syij = syij - (int)floor(syij + 0.5);
            szij = szij - (int)floor(szij + 0.5);

      /* squared distance between atom i and j */
            rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);

      /* Add force and energy contribution if distance between atoms smaller than rcut */
            if (rij_sq < rcut_sq) {
        rij = sqrt(rij_sq);
        dUpair_dr = splineEvalDiff(rij, pair_potential, PAIR_POTENTIAL_ROWS);
                drho_dr = splineEvalDiff(rij, electron_density, ELECTRON_DENSITY_ROWS);

        /* Add virial contribution from i-j interaction */
                force = -(dUpair_dr + (dUembed_drho[i] + dUembed_drho[j])*drho_dr) / rij;

        virial += force * rij_sq;
            }
    }
  }

  virial /= 3.0;

  free(sx); free(sy); free(sz); sx = NULL; sy = NULL; sz = NULL;
  free(density); density = NULL;
  free(dUembed_drho); dUembed_drho = NULL;

  return(virial);

}

double get_kinetic_AL(double velocities[][3], int nbr_of_dimensions, int nbr_atoms, double m_AL)
{
    double energy = 0;
    for (int j = 0; j < nbr_atoms; j++) {
        for (int k = 0; k < nbr_of_dimensions; k++) {
            energy += m_AL * pow(velocities[j][k], 2) / 2.0;
        }
    }
    return energy;
}
```

```
451 /* Calculation of instantaneous temperature, se 5.2 in molecular dynamics*/
452 double instantaneous_temperature(double kinetic_energy, int nbr_of_particles)
453 {
454     double temperature = 0;
455     temperature = 2.0/(k_b*nbr_of_particles*3) * kinetic_energy;
456     return temperature;
457 }
458
459 /* Calculation of temperature based on averaged kinetic energy */
460 double averaged_temperature(double* kinetic_energy, int nbr_of_particles, int ↩
        current_nbr_of_timesteps)
461 {
462     double temperature = 0;
463     double factor = 2.0/(3.0*k_b*nbr_of_particles*(current_nbr_of_timesteps+1.0)↩
          );
464     for (int i = 0; i < current_nbr_of_timesteps+1; i++)
465     {
466         temperature += kinetic_energy[i];
467     }
468     temperature*=factor;
469     return temperature;
470 }
471
472
473 /* Calculation of instantaneous pressure, se 5.3 in molecular dynamics*/
474 double instantaneous_pressure(double virial, double temperature, int ↩
        nbr_of_particles, double volume)
475 {
476     //double pressure = 0;
477     return (virial + temperature *k_b*nbr_of_particles) / volume;
478 }
479
480 /* Calculation of pressure based on averaged virial */
481 double averaged_pressure(double* virial, double* kinetic_energy, double volume, ↩
        int current_nbr_of_timesteps)
482 {
483     double pressure = 0;
484     for (int i = 0; i < current_nbr_of_timesteps+1; i++)
485     {
486         pressure += (virial[i] + 2.0/3.0*kinetic_energy[i]);
487
488     }
489     pressure /= volume*(current_nbr_of_timesteps+1.0);
490     return pressure;
491 }
```