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

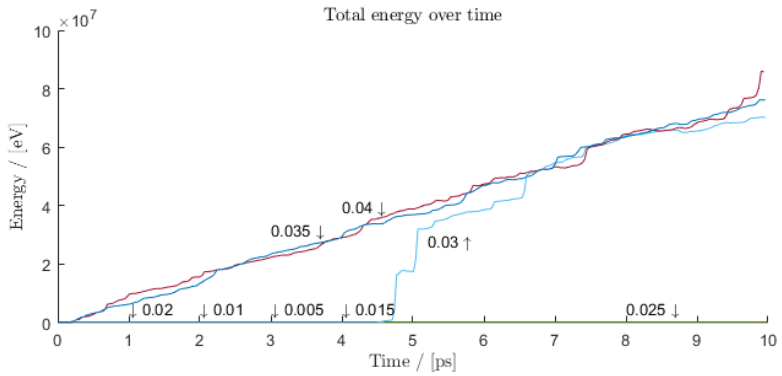

Figure 1: For different time-steps the energy evolves differently over time, in the figure we there are four simulations with different time-steps. The time-steps increases with 0.01 ps for each simulation and the total energy starts to increase for time-steps of 0.03 ps.

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

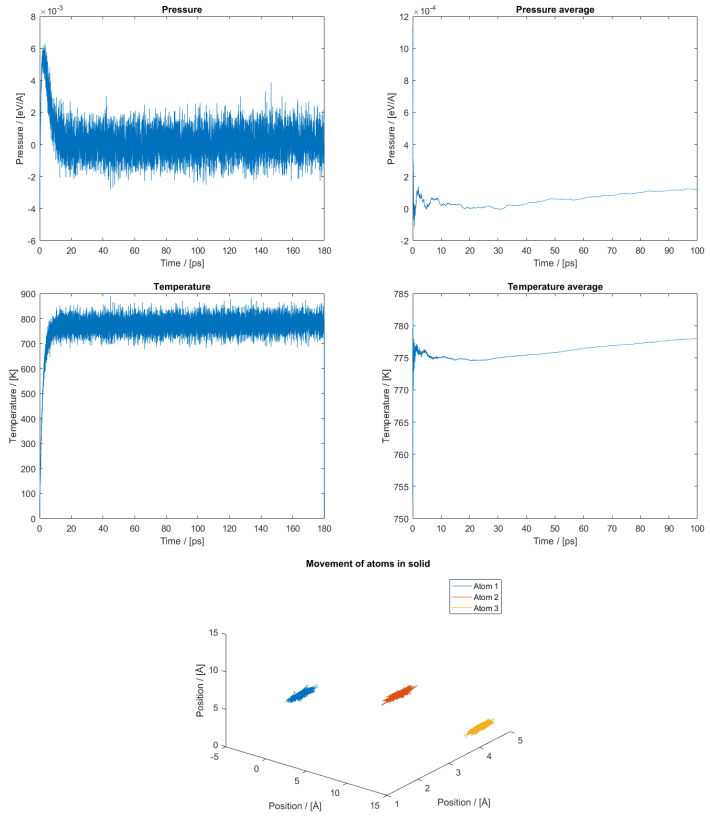

Figure 2: After the equilibrium, both the pressure and the temperature needs some time to stabilize after rescaling the velocities, but remains stable for longer time. The equalisation temperature was set to 500 C°.

The third problem was about implementing routines for equilibrating the molecular dynamics system to a specified temperature, $T_{eq} = 500°$ C, at pressure $P_{eq} = 1$ atm. The equilibration was implemented using scaling of the velocities and the total volume, and consquently the positions of the molecules as well. The equations used can be found in appendix D in the molecular dynamics lecture notes [1]. The goal was to study the temperature and pressure after the equilibration process through constant energy and volume simulation. We also plot the trajectories of a few particles to show that the system is still in a solid state.

The three main parameters for the equilibration are the timestep (used in the velocity Verlet algorithm) Δt and the temperature and pressure relaxation times, τ_T and τ_P respectively. The timestep used was 5 fs and the temperature relaxation time was chosen to be $\tau_T = 100\Delta t$, i.e. choosing the $\Delta t/\tau_T$ quotient in the α_T calculation to be equal 0.01.

When equilibrating the pressure the isothermic compressibility, κ_T, is used when computing $\alpha_P(t)$. The isothermic compressibility for aluminium is 0.01385 GPa^{-1}, but τ_P is chosen in such a way that the quotient $\kappa_T \Delta t/\tau_P$.

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

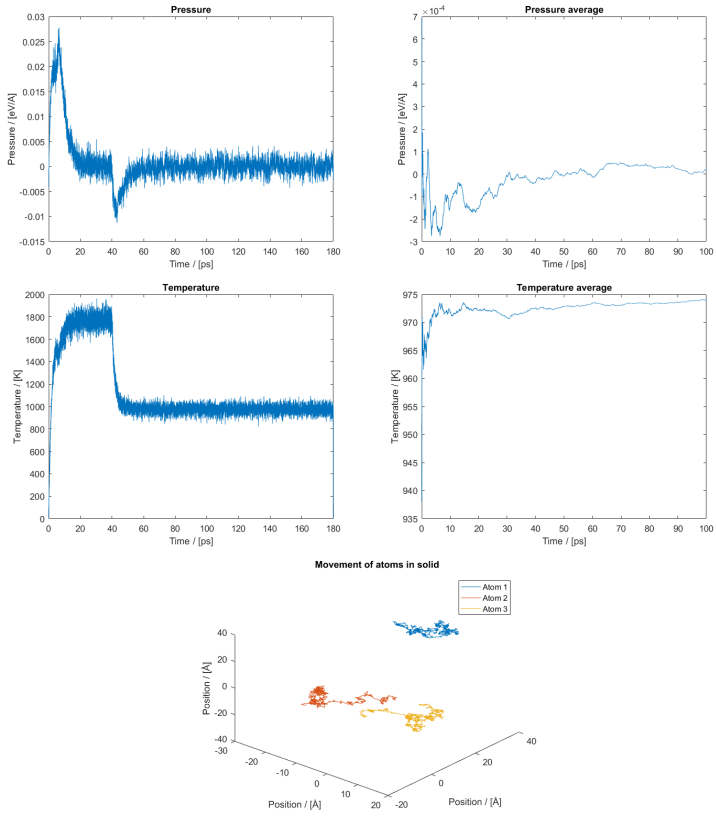

Figure 3: After the equilibrium, both the pressure and the temperature needs some time to stabilize after rescaling the velocities, but remains stable for longer time. The equalisation temperature was first set to 1000 C° for the smelting and then the temperature was reduced to 700 C°.

Problem 5

One interesting parameter of a material is the heat capacity. We can calculate it from fluctuations in either the kinetic or the potential energies,

$$C_v = \frac{3Nk_B}{2}\left[1 - \frac{2}{3Nk_b^2T^2}\left\langle(\delta\epsilon_{kin})^2\right\rangle_{NVE}\right]^{-1}, \tag{1}$$

$$C_v = \frac{3Nk_B}{2}\left[1 - \frac{2}{3Nk_b^2T^2}\left\langle\left(\delta\epsilon_{pot}\right)^2\right\rangle_{NVE}\right]^{-1}. \tag{2}$$

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)	$4.9634 \cdot 10^{-2}$	$6.7474 \cdot 10^{-2}$
$C_V/(\mathrm{eV/kg\,K})$ (potential)	$4.9686 \cdot 10^{-2}$	$6.7579 \cdot 10^{-2}$

Problem 6

When instead using the relation

$$C_V = \left(\frac{\partial E}{\partial T}\right)_{N,V} \tag{3}$$

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

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

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

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 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.

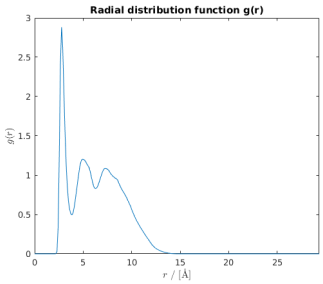

Figure 4: The radial function is computed by taking the histogram over all the internal distances between the atoms then divided by the random distribution of the same density.

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.

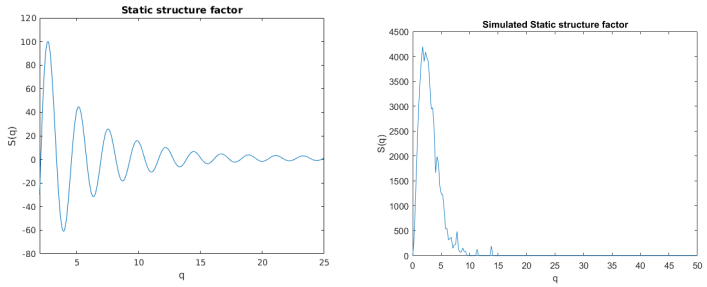

Figure 5: The static structure function computed both from the radial distribution function in figure 4 in the left and using bins in Fourier space to the right.

Concluding discussion

References

[1] Göran Wahnström. Molecular dynamics lecture notes. http://fy.chalmers.se/~tfsgw/CompPhys/lectures/MD_LectureNotes_151110.pdf.

A Source code

A.1 Task1/MD_main.c

```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"
#define nbr_of_particles 256
#define nbr_of_timesteps 1e4
#define nbr_of_timesteps_eq 4000
```

```c
#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
                * ((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);
                }
            }

        }
    }
```

```
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],
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++) {
```

```
                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];
        }
    }

    // Compute energies, temperature etc. at equilibrium
    energy[0] = get_energy_AL(q, cell_length, nbr_of_particles);
```

```
    virial[0] = get_virial_AL(q, cell_length, nbr_of_particles);
    energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL);
    temperature_avg[0] = instantaneous_temperature(energy_kin[0], nbr_of_particles);
    pressure_avg[0] = instantaneous_pressure(virial[0], temperature_avg[0],
        nbr_of_particles, volume);

    /* 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[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);

        // Temperature
        temperature_avg[i] = averaged_temperature(energy_kin, nbr_of_particles, i);
        temperature[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_temperature(energy_kin[i],
            nbr_of_particles);


        // 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

    /* Save data to file*/
    file = fopen("displacement.dat","w");

    double current_time;
    for (int i = 0; i < nbr_of_timesteps; i ++) {
        current_time = i*timestep;
        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.4 Task5/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 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[] = { 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 4↩
                                                x4x4 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];
        }
    }

    // Compute energies, temperature etc. at equilibrium
    energy_pot[0] = get_energy_AL(q, cell_length, nbr_of_particles);
    virial[0] = get_virial_AL(q, cell_length, nbr_of_particles);
```

```
    energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL);
    temperature_avg[0] = instantaneous_temperature(energy_kin[0], nbr_of_particles);
    pressure_avg[0] = instantaneous_pressure(virial[0], temperature_avg[0],
        nbr_of_particles, volume);

    /* 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);

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

        // Temperature
        temperature_avg[i] = averaged_temperature(energy_kin, nbr_of_particles,
            i);
        /*temperature[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_temperature
            (energy_kin[i],
            nbr_of_particles);*/


        // 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

    // Compute heat capacity
    heat_capacity_kin = calculate_heat_capacity_kin(energy_kin, temperature_eq
        [1],
        nbr_of_particles, nbr_of_timesteps);
    heat_capacity_pot = calculate_heat_capacity_pot(energy_pot, temperature_eq
        [1],
        nbr_of_particles, nbr_of_timesteps);

    printf("Temp: %f\nHeat capacity: %.10f \t %.10f\n", temperature_eq[1],
        heat_capacity_kin, heat_capacity_pot);

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



    free(energy_kin);       energy_kin = NULL;
    free(energy_pot);       energy_pot = 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.5 Task6/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 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[] = { 1000.0+273.15, 700.0+273.15 };
    double delta_temperature[] = { -10.0, 10.0 };
    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

    double heat_capacity;
    double energy_avg[2] = { 0 };
    double temperature_avg[2] = { 0 };


    /* Allocate memory for large vectors */

    double* energy_pot      = (double*) malloc(nbr_of_timesteps * sizeof(double));
    double* energy_kin      = (double*) malloc(nbr_of_timesteps * sizeof(double));


    /* 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);


```

```
71
72      /* Put atoms on lattice */
73      init_fcc(q, 4, lattice_param);
74
75
76      /* Initial conditions */
77      for (int i = 0; i < nbr_of_particles; i++){
78          for (int j = 0; j < nbr_of_dimensions; j++){
79
80              // Initial perturbation from equilibrium
81              q[i][j] += lattice_spacing * initial_displacement
82                  * ((double)rand()/(double)RAND_MAX);
83
84          }
85      }
86
87
88      get_forces_AL(f, q, cell_length, nbr_of_particles);
89
90      /* Simulation */
91      /* Equilibrium stage */
92
93      double inst_temperature_eq;
94      double inst_pressure_eq;
95      double alpha_T = 1.0;
96      double alpha_P = 1.0;
97      double energy_kin_eq = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,↩
            m_AL);
98      double virial_eq = get_virial_AL(q,cell_length,nbr_of_particles);
99
100     for (int d = 0; d < 2; d++) {
101
102         for (int equil = 0; equil < 2; equil++) {
103
104             double target_temp = temperature_eq[equil] + delta_temperature[d];
105
106             for (int i = 1; i < nbr_of_timesteps_eq; i++)
107             {
108                 /** Verlet algorithm **/
109                 /* Half step for velocity */
110                 for (int j = 0; j < nbr_of_particles; j++){
111                     for (int k = 0; k < nbr_of_dimensions; k++){
112                         v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
113                     }
114                 }
115
116                 /* Update displacement*/
117                 for (int j = 0; j < nbr_of_particles; j++){
118                     for (int k = 0; k < nbr_of_dimensions; k++){
119                         q[j][k] += timestep * v[j][k];
120                     }
121                 }
122
123                 /* Forces */
124                 get_forces_AL(f,q,cell_length,nbr_of_particles);
125
126                 /* Final velocity*/
127                 for (int j = 0; j < nbr_of_particles; j++){
128                     for (int k = 0; k < nbr_of_dimensions; k++){
129                         v[j][k] += timestep * 0.5* f[j][k]/m_AL;
130                     }
131                 }
132
133                 /* Calculate energy */
134                 // Kinetic energy
135                 energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, ↩
                        nbr_of_particles, m_AL);
136
137                 virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);
138
139
140                 inst_temperature_eq = instantaneous_temperature(energy_kin_eq, ↩
                        nbr_of_particles);
141                 inst_pressure_eq = instantaneous_pressure(virial_eq, ↩
                        inst_temperature_eq,
142                     nbr_of_particles, volume);
143
144
145                 // Update alhpas
146                 alpha_T = 1.0 + 0.01*(target_temp-inst_temperature_eq)/↩
                        inst_temperature_eq;
147                 alpha_P = 1.0 - 0.01*(pressure_eq - inst_pressure_eq);
148
149                 // Scale velocities
150                 for (int j = 0; j < nbr_of_particles; j++){
151                     for (int k = 0; k < nbr_of_dimensions; k++){
152                         v[j][k] *= sqrt(alpha_T);
153                     }
154                 }
155
156                 // 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 4↩
                                                x4x4 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
164             // Scale velocities
165             for (int j = 0; j < nbr_of_particles; j++) {
166                 for (int k = 0; k < nbr_of_dimensions; k++) {
167                     v[j][k] *= sqrt(alpha_T);
168                 }
169             }
170
171             // Scale positions and volume
172             cell_length *= pow(alpha_P, 1.0/3.0);
173             volume = pow(cell_length, 3);
174             for (int j = 0; j < nbr_of_particles; j++) {
175                 for (int k = 0; k < nbr_of_dimensions; k++) {
176                     q[j][k] *= pow(alpha_P, 1.0/3.0);
177                 }
178             }
179
180         }
181     }
182
183     for (int i = 0; i < nbr_of_particles; i++) {
184         for (int j = 0; j < nbr_of_dimensions; j++) {
185             qq(0,i,j)=q[i][j];
186         }
187     }
188
189
190     // Compute energies, temperature etc. at equilibrium
191     double min = 0.0;
192     double max = sqrt(3*cell_length*cell_length);
193     double d_r = (max-min)/(1.0*k_bins);
194     int bins[k_bins];
195     int* bins2 = (int*) malloc(k_bins * sizeof(int));
196
197     for (int i = 0; i < k_bins; i++) {
198         bins[i]=0;
199         bins2[i]=0;
200     }
201
202     for (int i = 1; i < nbr_of_timesteps; i++)
203     {
204         /** Verlet algorithm **/
205         /* Half step for velocity */
206         for (int j = 0; j < nbr_of_particles; j++){
207             for (int k = 0; k < nbr_of_dimensions; k++){
208                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
209             }
210         }
211
212         /* Update displacement*/
213         for (int j = 0; j < nbr_of_particles; j++){
214             for (int k = 0; k < nbr_of_dimensions; k++){
215                 q[j][k] += timestep * v[j][k];
216             }
217         }
218
219         /* Forces */
220         get_forces_AL(f,q,cell_length,nbr_of_particles);
221
222         /* Final velocity*/
223         for (int j = 0; j < nbr_of_particles; j++){
224             for (int k = 0; k < nbr_of_dimensions; k++){
225                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
226             }
227         }
228
229         /* Calculate energy */
230         // Potential energy
231         energy[i] = get_energy_AL(q,cell_length,nbr_of_particles);
232         // Kinetic energy
233         energy_kin[i] = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,m_AL↩
                );
234
235         virial[i]=get_virial_AL(q,cell_length,nbr_of_particles);
236
237         /* Save current displacements to array*/
238         for (int j = 0; j < nbr_of_particles; j++){
239             for (int k = 0; k < nbr_of_dimensions; k++){
240                 qq(i,j,k)=q[j][k];
241             }
242         }
243     }
244
```

```
245
246     // Create Histogram
247
248     for (int i = 1; i < nbr_of_timesteps; i++)
249     {
250         for (int j = 1 ; j < nbr_of_particles; j++) {
251             for (int k = j+1 ; k < nbr_of_particles; k++) {
252
253                 double q1[nbr_of_dimensions];
254                 double q2[nbr_of_dimensions];
255                 for (int d = 0; d < nbr_of_dimensions; d++) {
256                     q1[d] = qq(i,j,d);
257                     q2[d] = qq(i,k,d);
258                 }
259                 double distance_sq = boundary_condition_dist_sq(q1, q2, ↩
                        cell_length);
260                 double dist = sqrt(distance_sq);
261                 int bin = get_bin(dist,min,max,d_r);
262                 bins2[bin] += 2;
263             }
264         }
265     }
266     double Nideal[k_bins];
267     double factor =((double)(nbr_of_particles-1.0))/volume * 4.0*PI/3.0;
268     for (int i = 0; i < k_bins; i++) {
269         Nideal[i] = factor*(3.0*i*i-3.0*i+1.0)*d_r*d_r*d_r;
270     }
271
272
273
274
275     /* Save data to file*/
276     file = fopen("histogram.dat","w");
277     for (int i = 0; i < k_bins; i ++) {
278         fprintf(file, "%e \t %i \t %i \t %e \n",d_r*(i-0.5), bins[i],bins2[i], ↩
                Nideal[i]);
279     }
280     fclose(file);
281     // TO THIS ISH TODO
282
283
284     free(energy_kin);       energy_kin = NULL;
285     free(energy);           energy = NULL;
286     free(disp_arr);         disp_arr = NULL;
287     free(virial);           virial = NULL;
288     free(temperature_avg);  temperature_avg = NULL;
289     free(pressure_avg);     pressure_avg = NULL;
290
291     return 0;
292 }
293
294 int get_bin(double val , double min , double max , double  d_r)
295 {
296     int bin = 0;
297     double current = min;
298     while (current <= val)
299     {
300         current += d_r;
301         bin++;
302     }
303     if (current > max)
304         return --bin;
305     return bin;
306 }
307
308 double boundary_condition_dist_sq(double u1[3], double u2[3], double L)
309 {
310     double d[3];
311     for (int i = 0; i < 3; i++) {
312         u1[i] /= L;
313         u2[i] /= L;
314
315         u1[i] -= floor(u1[i]);
316         d[i] = u1[i] - (u2[i] - floor(u2[i]));
317         d[i] -= (double)((int)floor(d[i]+0.5));
318     }
319
320     double sum = 0.0;
321     for (int i = 0; i < 3; i++)
322         sum += pow(d[i], 2);
323     return L*L * sum;
324
325 }
```

A.7 Task8/MD_main.c

```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 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);
                }
            }

        }
    }

    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
    energy[0] = get_energy_AL(q, cell_length, nbr_of_particles);
    virial[0] = get_virial_AL(q, cell_length, nbr_of_particles);
    energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL);
    temperature_avg[0] = instantaneous_temperature(energy_kin[0], nbr_of_particles);
    pressure_avg[0] = instantaneous_pressure(virial[0], temperature_avg[0],
        nbr_of_particles, volume);

    /* 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[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);

        // Temperature
        temperature_avg[i] = averaged_temperature(energy_kin, nbr_of_particles, i);
        temperature[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_temperature(energy_kin[i],
            nbr_of_particles);


        // 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));

  double *density = malloc(nbr_atoms * sizeof (double));
  double *dUembed_drho = malloc(nbr_atoms * sizeof (double));

  rcut = 6.06;
  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 */
```

```
            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 force contribution from i-j interaction */
                force = -(dUpair_dr + (dUembed_drho[i] + dUembed_drho[j])*↩
                     drho_dr) / rij;
                fx[i] += force * sxij * cell_length;
                fy[i] += force * syij * cell_length;
                fz[i] += force * szij * cell_length;
                fx[j] -= force * sxij * cell_length;
                fy[j] -= force * syij * cell_length;
                fz[j] -= force * szij * cell_length;
            }
    }
  }

  for (i = 0; i < nbr_atoms; i++){
    forces[i][0] = fx[i];
    forces[i][1] = fy[i];
    forces[i][2] = fz[i];
  }

  free(sx); free(sy); free(sz); sx = NULL; sy = NULL; sz = NULL;
  free(fx); free(fy); free(fz); fx = NULL; fy = NULL; fz = NULL;
  free(density); density = NULL;
  free(dUembed_drho); dUembed_drho = NULL;

}

/* Returns the potential energy */
double get_energy_AL(double positions[][3], double cell_length, int nbr_atoms)
{
  int i, j;
  double cell_length_inv, cell_length_sq;
  double rcut, rcut_sq;
  double energy;
  double densityi, dens;
  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 *density = malloc(nbr_atoms * sizeof (double));

  rcut = 6.06;
  rcut_sq = rcut * rcut;

  cell_length_inv = 1 / cell_length;
  cell_length_sq = cell_length * cell_length;

  for (i = 0; i < nbr_atoms; i++){
    sx[i] = positions[i][0] * cell_length_inv;
    sy[i] = positions[i][1] * cell_length_inv;
    sz[i] = positions[i][2] * cell_length_inv;
  }

  for (i = 0; i < nbr_atoms; i++){
    density[i] = 0;
  }

  energy = 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);

```

```
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 */
368             rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
369 
370       /* 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;
}


/* Calculation of instantaneous temperature, se 5.2 in molecular dynamics*/
double instantaneous_temperature(double kinetic_energy, int nbr_of_particles)
{
    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 }
```