

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 N ^o	Points	Avail. points
Σ		

Introduction

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

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

Problem 1

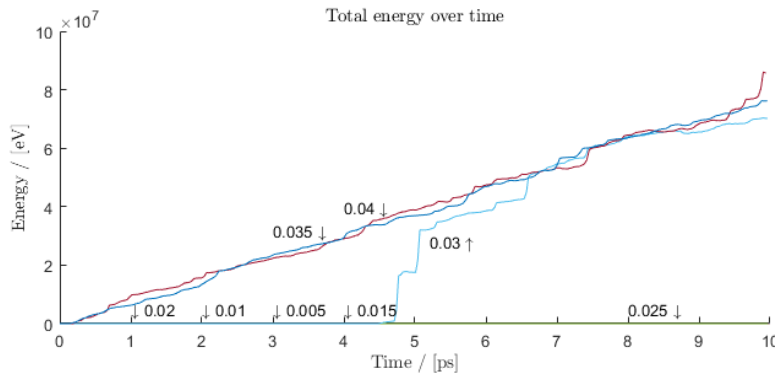


Figure 1: For different time-steps the energy evolves differently over time, in the figure we there are four simulations with different time-steps. The time-steps increases with 0.01 ps for each simulation and the total energy starts to increase for time-steps of 0.03 ps.

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

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

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

Problem 3

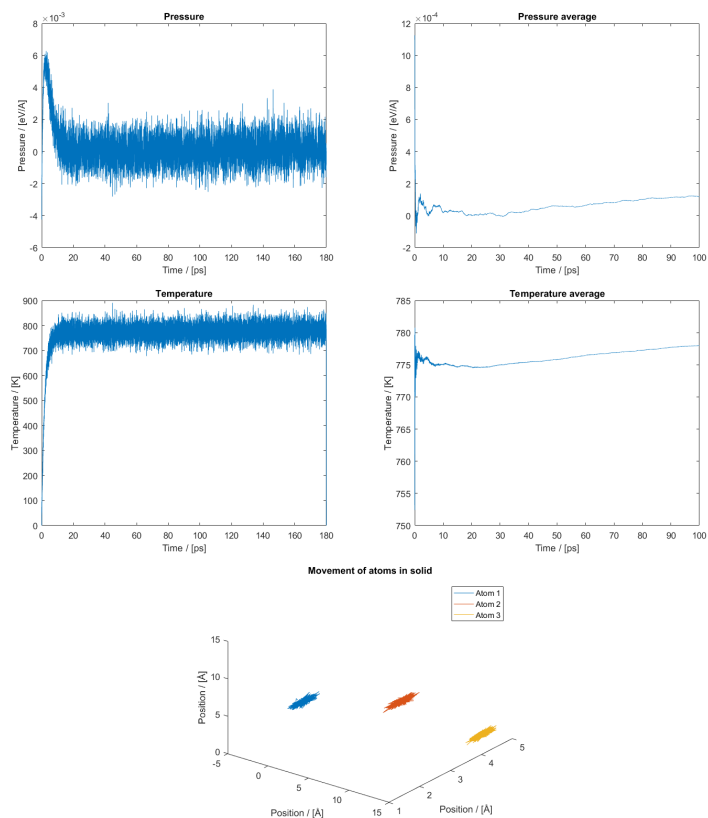


Figure 2: After the equilibrium, both the pressure and the temperature needs some time to stabilize after rescaling the velocities, but remains stable for longer time. The equalisation temperature was set to 500 C° .

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

Problem 4

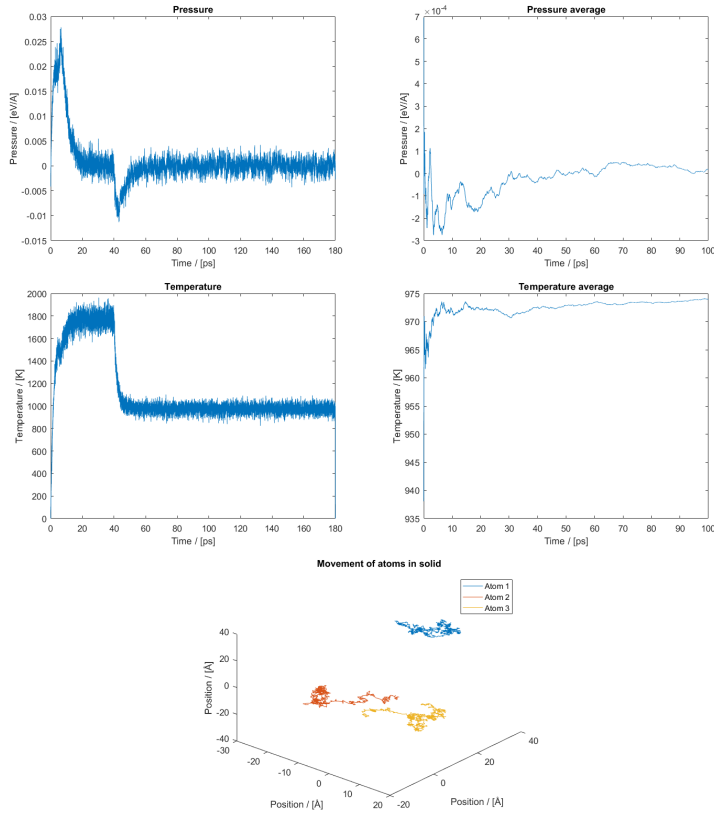


Figure 3: After the equilibrium, both the pressure and the temperature needs some time to stabilize after rescaling the velocities, but remains stable for longer time. The equalisation temperature was first set to 1000 C° for the smelting and then the temperature was reduced to 700 C°.

Problem 5

$$1 \frac{\text{J}}{\text{molK}} = 1.0366e-05 \frac{\text{eV}}{\text{K}}$$

$$C_V [AL] : 24.20 \frac{\text{J}}{\text{molK}} = 1.0366e-05 \frac{\text{eV}}{\text{K}}$$

From our MD simulations we obtained the following values for C_V when using the potential and kinetic energy fluctuations respectively.

Table 1: Heat capacity obtained by measuring energy fluctuations.

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

Problem 6

When instead using the relation

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

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

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

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

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

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

Problem 7

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

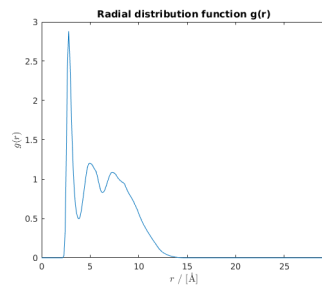


Figure 4: The radial function is computed by taking the histogram over all the internal distances between the atoms then divided by the random distribution of the same density.

Problem 8

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

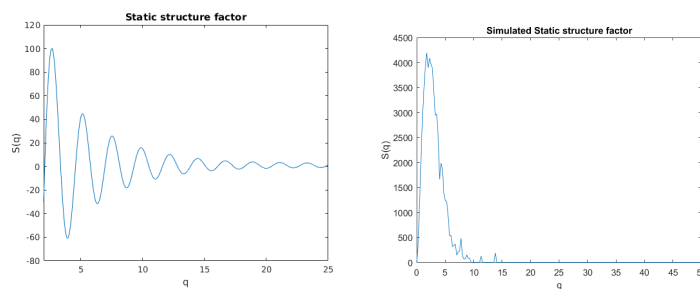


Figure 5: The static structure function computed both from the radial distribution function in figure 4 in the left and using bins in Fourier space to the right.

Concluding discussion

A Source code

A.1 Task1/MD_main.c

```
1  /*
2  MD_main.c
3
4  Created by Anders Lindman on 2013-10-31.
5  */
6
7  #include <stdio.h>
8  #include <math.h>
9  #include <stdlib.h>
10 #include <time.h>
11 #include <string.h>
12 #include "initfcc.h"
13 #include "alpotential.h"
14 #define nbr_of_particles 256
15 #define nbr_of_dimensions 3
16
17 /* Main program */
18 int main()
19 {
20     srand(time(NULL));
21
22     /* Simulation parameters */
23     double m_AL; // Mass of atom
24     double cell_length; // Side length of supercell
25
26     double lattice_spacing; // Smallest length between atoms
27     double initial_displacement; // Initial displacement of the atoms from ←
28         // lattice positions
29
30     double lattice_param; // Lattice parameter, length of each side in the
31         // unit cell
32     double timesteps[8];
33
34     FILE *file;
35
36     int time_length = 10;
37
38     /* Initialize parameters*/
39     initial_displacement = 0.05;
40     lattice_param = 4.046; // For aluminium ( )
41     lattice_spacing = lattice_param/sqrt(2.0);
42     //timestep = 0.01; // 0.1 Bad, 0.01 Seems decent
43
44     m_AL = 0.0027964; // In ASU
45     cell_length = 4*lattice_param; // Side of the supercell: The 256 atoms are
46         // structured in a block of 4x4x4 unit cells
47
48     // Test different timestep with 0.01 difference
49     for (int i = 0; i < 8; i++)
50         timesteps[i]=0.005*(i+1);
51
52     for (int t = 0; t < 8; t++)
53     {
54         // Current timestep and number of timesteps
55         double timestep = timesteps[t];
56         int nbr_of_timesteps = (int)(time_length/timestep);
57
58         /* Current displacement, velocities, and acceleratons */
59         double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
60         double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
61         double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces
62
63
64
65         /* Allocate memory for large vectors */
66         /* Simulate 3 dimensional data by placing initializeing a 1-dimensional ←
67             array*/
68         double* energy_pot =(double*)malloc(nbr_of_timesteps*sizeof(double));
69         double* energy_kin = (double*)malloc(nbr_of_timesteps*sizeof(double));
70
71         /* Put atoms on lattice */
72         init_fcc(q, 4, lattice_param);
73
74         /* Initial conditions */
75         for (int i = 0; i < nbr_of_particles; i++){
76             for (int j = 0; j < nbr_of_dimensions; j++){
77
78                 // Initial perturbation from equilibrium
```

```

79         q[i][j] +=lattice_spacing* initial_displacement
80         * ((double)rand()/((double)RAND_MAX));
81     }
82 }
83 energy_pot[0]=get_energy_AL(q,cell_length,nbr_of_particles);
84 energy_kin[0]=get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,m_AL);
85
86 get_forces_AL(f,q,cell_length,nbr_of_particles);
87
88
89 /* Simulation */
90 for (int i = 1; i < nbr_of_timesteps; i++)
91 {
92     /** Verlet algorithm */
93     /* Half step for velocity */
94     for (int j = 0; j < nbr_of_particles; j++){
95         for (int k = 0; k < nbr_of_dimensions; k++){
96             v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
97         }
98     }
99
100     /* Update displacement*/
101     for (int j = 0; j < nbr_of_particles; j++){
102         for (int k = 0; k < nbr_of_dimensions; k++){
103             q[j][k] += timestep * v[j][k];
104         }
105     }
106
107     /* Forces */
108     get_forces_AL(f,q,cell_length,nbr_of_particles);
109
110     /* Final velocity*/
111     for (int j = 0; j < nbr_of_particles; j++){
112         for (int k = 0; k < nbr_of_dimensions; k++){
113             v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
114         }
115     }
116
117     /* Calculate energy */
118     // Potential energy
119     energy_pot[i] = get_energy_AL(q,cell_length,nbr_of_particles);
120     // Kinetic energy
121     energy_kin[i] = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,m_AL);
122 }
123 char str[80];
124 char S[3];
125 sprintf(S, "%.3f", timestep);
126
127 strcpy (str,"data/energy");
128 strcat (str,S);
129 strcat (str,".dat");
130
131
132 /* Save energies to file */
133 file = fopen(str,"w");
134
135 double current_time;
136 for (int i = 0; i < nbr_of_timesteps; i ++){
137     {
138         current_time = i*timestep;
139         fprintf(file, "%.4f \t", current_time);
140         fprintf(file, "%.4f \t", energy_pot[i]);
141         fprintf(file, "%.4f \n", energy_kin[i]);
142     }
143     fclose(file);
144
145
146     free(energy_kin); energy_kin=NULL;
147     free(energy_pot); energy_pot=NULL;
148 }
149
150 return 0;
151 }
152

```

A.2 Task3/MD_main.c

```

1
2
3 #include <stdio.h>
4 #include <math.h>
5 #include <stdlib.h>
6 #include <time.h>
7 #include "initfcc.h"
8 #include "alpotential.h"

```

```

9  #define nbr_of_particles 256
10 #define nbr_of_timesteps 1e4
11 #define nbr_of_timesteps_eq 4000
12 #define nbr_of_dimensions 3
13
14 double boundary_condition(double,double);
15
16
17
18 /* Main program */
19 int main()
20 {
21     srand(time(NULL));
22
23     /* Simulation parameters */
24     double m_AL; // Mass of atom
25     double cell_length; // Side length of supercell
26     double volume;
27     double lattice_spacing; // Smallest length between atoms
28     double initial_displacement; // Initial displacement of the atoms from ←
        their
        // lattice positions
29
30     double lattice_param; // Lattice parameter, length of each side in the
        // unit cell
31
32     double timestep;
33     double temperature_eq[] = { 500.0+273.15, 500.0+273.15 };
34     double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
35
36     FILE *file;
37
38
39     /* Current displacement, velocities, and acceleratons */
40     double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
41     double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
42     double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces
43
44     /* Allocate memory for large vectors */
45     /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional ←
        array*/
46     #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+←
        nbr_of_dimensions*j+k])
47     double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*←
        nbr_of_dimensions*sizeof(double));
48
49     double* energy = (double*) malloc(nbr_of_timesteps * sizeof(double)←
        );
50     double* energy_kin = (double*) malloc(nbr_of_timesteps * sizeof(double)←
        );
51     double* virial = (double*) malloc(nbr_of_timesteps * sizeof(double)←
        );
52     double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)←
        );
53     double* pressure_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)←
        );
54     double* temperature = (double*) malloc((2 * nbr_of_timesteps_eq + ←
        nbr_of_timesteps) * sizeof(double));
55     double* pressure = (double*) malloc((2 * nbr_of_timesteps_eq + ←
        nbr_of_timesteps) * sizeof(double));
56
57     //TODO go over parameters again
58     /* Initialize parameters*/
59     initial_displacement = 0.05;
60     lattice_param = 4.046; // For aluminium ( )
61     lattice_spacing = lattice_param/sqrt(2.0);
62     timestep = 0.01; // 0.1 Bad, 0.01 Seems decent
63     m_AL = 0.0027964; // In ASU
64     cell_length = 4*lattice_param; // Side of the supercell: The ←
        256 atoms are
        // structured in a block of 4←
        x4x4 unit cells
65
66     volume = pow(cell_length, 3);
67
68     // Initialize all displacements, for all times, as 0
69     for (int i = 0; i < nbr_of_timesteps; i++){
70         for (int j = 0; j < nbr_of_particles; j++){
71             for (int k = 0; k < nbr_of_dimensions; k++){
72                 qq(i,j,k) = 0;
73             }
74         }
75     }
76
77     /* Put atoms on lattice */
78     init_fcc(q, 4, lattice_param);
79
80
81     /* Initial conditions */
82     for (int i = 0; i < nbr_of_particles; i++){
83         for (int j = 0; j < nbr_of_dimensions; j++){
84
85             // Initial perturbation from equilibrium
86             q[i][j] += lattice_spacing * initial_displacement

```



```

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

```

```

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

```

```

255     fprintf(file, "%.4f \t", current_time );
256     for (int j = 0; j < nbr_of_particles; j++) {
257         for (int k = 0; k < nbr_of_dimensions; k++) {
258             fprintf(file, "%.4f \t", qq(i,j,k));
259         }
260     }
261     fprintf(file, "\n");
262 }
263 fclose(file);
264
265 /* Save energies to file */
266 file = fopen("energy.dat", "w");
267
268 for (int i = 0; i < nbr_of_timesteps; i++) {
269     current_time = i*timestep;
270     fprintf(file, "%.4f \t", current_time);
271     fprintf(file, "%.4f \t", energy[i]);
272     fprintf(file, "%.4f \n", energy_kin[i]);
273 }
274 fclose(file);
275
276 // Save temperature to file
277 file = fopen("temperature.dat", "w");
278 for (int i = 0; i < 2*nbr_of_timesteps_eq+nbr_of_timesteps; i++) {
279     current_time = i*timestep;
280     fprintf(file, "%.3f \t %e\n", current_time, temperature[i]);
281 }
282 fclose(file);
283
284 file = fopen("temperature_avg.dat", "w");
285 for (int i = 0; i < nbr_of_timesteps; i++) {
286     current_time = i*timestep;
287     fprintf(file, "%.3f \t %e\n", current_time, temperature_avg[i]);
288 }
289 fclose(file);
290
291 // Save pressure to file
292 file = fopen("pressure.dat", "w");
293 for (int i = 0; i < 2*nbr_of_timesteps_eq+nbr_of_timesteps; i++) {
294     current_time = i*timestep;
295     fprintf(file, "%.3f \t %e \n", current_time, pressure[i]);
296 }
297 fclose(file);
298
299 file = fopen("pressure_avg.dat", "w");
300 for (int i = 0; i < nbr_of_timesteps; i++) {
301     current_time = i*timestep;
302     fprintf(file, "%.3f \t %e\n", current_time, pressure_avg[i]);
303 }
304 fclose(file);
305
306 free(energy_kin);      energy_kin = NULL;
307 free(energy);          energy = NULL;
308 free(displacement);    displacement = NULL;
309 free(virial);          virial = NULL;
310 free(temperature_avg); temperature_avg = NULL;
311 free(pressure_avg);    pressure_avg = NULL;
312 free(temperature);     temperature = NULL;
313 free(pressure);        pressure = NULL;
314
315 return 0;
316 }
317

```

A.3 Task4/MD main.c

```

1
2
3 #include <stdio.h>
4 #include <math.h>
5 #include <stdlib.h>
6 #include <time.h>
7 #include "initfcc.h"
8 #include "alpotential.h"
9 #define nbr_of_particles 256
10 #define nbr_of_timesteps 1e4
11 #define nbr_of_timesteps_eq 4000
12 #define nbr_of_dimensions 3
13
14 double boundary_condition(double, double);
15
16
17
18 /* Main program */
19 int main()
20 {

```

```

21 srand(time(NULL));
22
23 /* Simulation parameters */
24 double m_AL; // Mass of atom
25 double cell_length; // Side length of supercell
26 double volume;
27 double lattice_spacing; // Smallest length between atoms
28 double initial_displacement; // Initial displacement of the atoms from ←
    their
    // lattice positions
29
30 double lattice_param; // Lattice parameter, length of each side in the
31 // unit cell
32
33 double timestep;
34 double temperature_eq[] = { 1500.0+273.15, 700.0+273.15 };
35 double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
36
37 FILE *file;
38
39 /* Current displacement, velocities, and acceleratons */
40 double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
41 double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
42 double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces
43
44 /* Allocate memory for large vectors */
45 /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional ←
    array*/
46 #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+←
    nbr_of_dimensions*j+k])
47 double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*←
    nbr_of_dimensions*sizeof(double));
48
49 double* energy = (double*) malloc(nbr_of_timesteps * sizeof(double)←
    );
50 double* energy_kin = (double*) malloc(nbr_of_timesteps * sizeof(double)←
    );
51 double* virial = (double*) malloc(nbr_of_timesteps * sizeof(double)←
    );
52 double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)←
    );
53 double* pressure_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)←
    );
54 double* temperature = (double*) malloc((2 * nbr_of_timesteps_eq + ←
    nbr_of_timesteps) * sizeof(double));
55 double* pressure = (double*) malloc((2 * nbr_of_timesteps_eq + ←
    nbr_of_timesteps) * sizeof(double));
56
57 //TODO go over parameters again
58 /* Initialize parameters*/
59 initial_displacement = 0.05;
60 lattice_param = 4.046; // For aluminium ( )
61 lattice_spacing = lattice_param/sqrt(2.0);
62 timestep = 0.01; // 0.1 Bad, 0.01 Seems decent
63 m_AL = 0.0027964; // In ASU
64 cell_length = 4*lattice_param; // Side of the supercell: The ←
    256 atoms are
    // structured in a block of 4←
    x4x4 unit cells
65
66 volume = pow(cell_length, 3);
67
68 // Initialize all displacements, for all times, as 0
69 for (int i = 0; i < nbr_of_timesteps; i++){
70     for (int j = 0; j < nbr_of_particles; j++){
71         for (int k = 0; k < nbr_of_dimensions; k++){
72             qq(i,j,k) = 0;
73         }
74     }
75 }
76
77 /* Put atoms on lattice */
78 init_fcc(q, 4, lattice_param);
79
80
81 /* Initial conditions */
82 for (int i = 0; i < nbr_of_particles; i++){
83     for (int j = 0; j < nbr_of_dimensions; j++){
84
85         // Initial perturbation from equilibrium
86         q[i][j] += lattice_spacing * initial_displacement
87             * ((double)rand()/((double)RAND_MAX);
88
89     }
90 }
91
92 get_forces_AL(f, q, cell_length, nbr_of_particles);
93
94 /* Simulation */
95 /* Equilibrium stage */
96
97 double inst_temperature_eq;

```

```

99 double inst_pressure_eq;
100 double alpha_T = 1.0;
101 double alpha_P = 1.0;
102 double energy_kin_eq = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,↵
    m_AL);
103 double virial_eq = get_virial_AL(q,cell_length,nbr_of_particles);
104
105 temperature[0] = instantaneous_temperature(energy_kin_eq, nbr_of_particles)↵
    ;
106 pressure[0] = instantaneous_pressure(virial_eq, temperature[0], ↵
    nbr_of_particles, volume);
107
108 for (int equil = 0; equil < 2; equil++) {
109     for (int i = 1; i < nbr_of_timesteps_eq; i++)
110     {
111         /** Verlet algorithm */
112         /* Half step for velocity */
113         for (int j = 0; j < nbr_of_particles; j++){
114             for (int k = 0; k < nbr_of_dimensions; k++){
115                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
116             }
117         }
118
119         /* Update displacement*/
120         for (int j = 0; j < nbr_of_particles; j++){
121             for (int k = 0; k < nbr_of_dimensions; k++){
122                 q[j][k] += timestep * v[j][k];
123             }
124         }
125
126         /* Forces */
127         get_forces_AL(f,q,cell_length,nbr_of_particles);
128
129         /* Final velocity*/
130         for (int j = 0; j < nbr_of_particles; j++){
131             for (int k = 0; k < nbr_of_dimensions; k++){
132                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
133             }
134         }
135
136         /* Calculate energy */
137         // Kinetic energy
138         energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, ↵
            nbr_of_particles, m_AL);
139
140         virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);
141
142
143         inst_temperature_eq = instantaneous_temperature(energy_kin_eq, ↵
            nbr_of_particles);
144         temperature[equil*(nbr_of_timesteps_eq-1) + i] = inst_temperature_eq↵
            ;
145         inst_pressure_eq = instantaneous_pressure(virial_eq, ↵
            inst_temperature_eq,
146             nbr_of_particles, volume);
147         pressure[equil*(nbr_of_timesteps_eq-1) + i] = inst_pressure_eq;
148
149         // Update alhpas
150         alpha_T = 1.0 + 0.01*(temperature_eq[equil]-inst_temperature_eq)/↵
            inst_temperature_eq;
151         alpha_P = 1.0 - 0.01*(pressure_eq - inst_pressure_eq);
152
153         // Scale velocities
154         for (int j = 0; j < nbr_of_particles; j++){
155             for (int k = 0; k < nbr_of_dimensions; k++){
156                 v[j][k] *= sqrt(alpha_T);
157             }
158         }
159
160         // Scale positions and volume
161         cell_length *= pow(alpha_P, 1.0/3.0);
162         volume = pow(cell_length, 3);
163         for (int j = 0; j < nbr_of_particles; j++) {
164             for (int k = 0; k < nbr_of_dimensions; k++) {
165                 q[j][k] *= pow(alpha_P, 1.0/3.0);
166             }
167         }
168     }
169 }
170
171 }
172 }
173
174 printf("Equilibration done.\n");
175 printf("Cell length: %.8f \n", cell_length);
176
177 for (int i = 0; i < nbr_of_particles; i++){
178     for (int j = 0; j < nbr_of_dimensions; j++){
179         qq(0,i,j)=q[i][j];
180     }
181 }

```

```

182
183 // Compute energies, temperature etc. at equilibrium
184 energy[0] = get_energy_AL(q, cell_length, nbr_of_particles);
185 virial[0] = get_virial_AL(q, cell_length, nbr_of_particles);
186 energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, m_AL)↵
187 ;
188 temperature_avg[0] = instantaneous_temperature(energy_kin[0], ↵
189     nbr_of_particles);
190 pressure_avg[0] = instantaneous_pressure(virial[0], temperature_avg[0],
191     nbr_of_particles, volume);
192
193 /* Simulation after equilibrium*/
194 for (int i = 1; i < nbr_of_timesteps; i++)
195 {
196     /** Verlet algorithm **/
197     /* Half step for velocity */
198     for (int j = 0; j < nbr_of_particles; j++){
199         for (int k = 0; k < nbr_of_dimensions; k++){
200             v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
201         }
202     }
203
204     /* Update displacement*/
205     for (int j = 0; j < nbr_of_particles; j++){
206         for (int k = 0; k < nbr_of_dimensions; k++){
207             q[j][k] += timestep * v[j][k];
208         }
209     }
210
211     /* Update Forces */
212     get_forces_AL(f, q, cell_length, nbr_of_particles);
213
214     /* Final velocity*/
215     for (int j = 0; j < nbr_of_particles; j++){
216         for (int k = 0; k < nbr_of_dimensions; k++){
217             v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
218         }
219     }
220
221     /* Calculate energy */
222     // Potential energy
223     energy[i] = get_energy_AL(q, cell_length, nbr_of_particles);
224     // Kinetic energy
225     energy_kin[i] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, ↵
226         m_AL);
227
228     virial[i] = get_virial_AL(q, cell_length, nbr_of_particles);
229
230     // Temperature
231     temperature_avg[i] = averaged_temperature(energy_kin, nbr_of_particles, ↵
232         i);
233     temperature[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_temperature(↵
234         energy_kin[i],
235         nbr_of_particles);
236
237     // Pressure
238     pressure_avg[i] = averaged_pressure(virial, energy_kin, volume, i);
239     pressure[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_pressure(virial[↵
240         i],
241         temperature[2*(nbr_of_timesteps_eq-1) + i],
242         nbr_of_particles, volume);
243
244     /* Save current displacements to array*/
245     for (int j = 0; j < nbr_of_particles; j++){
246         for (int k = 0; k < nbr_of_dimensions; k++){
247             qq(i,j,k)=q[j][k];
248         }
249     }
250
251 } // equilibration/simulation
252
253 /* Save data to file*/
254 file = fopen("displacement.dat","w");
255
256 double current_time;
257 for (int i = 0; i < nbr_of_timesteps; i++) {
258     current_time = i*timestep;
259     fprintf(file, "%.4f \t", current_time );
260     for (int j = 0; j < nbr_of_particles; j++) {
261         for (int k = 0; k < nbr_of_dimensions; k++) {
262             fprintf(file, "%.4f \t", qq(i,j,k));
263         }
264     }
265     fprintf(file, "\n");
266 }
267 fclose(file);
268
269 /* Save energies to file */
270 file = fopen("energy.dat","w");

```

```

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

```

A.4 Task5/MD_main.c

```

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

```

```

32 double initial_displacement; // Initial displacement of the atoms from
    their
33 // lattice positions
34 double lattice_param; // Lattice parameter, length of each side in the
35 // unit cell
36 double timestep;
37 double temperature_eq[] = { 1000.0+273.15, 700.0+273.15 };
38 double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
39 double isothermal_compressibility = 1.0; //0.8645443196; // 1.385e-11 m^2/N
    = 1.385/1.602 ^3/eV
40
41 FILE *file;
42
43
44 /* Current displacement, velocities, and acceleratons */
45 double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
46 double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
47 double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces
48
49 double heat_capacity_pot, heat_capacity_kin;
50
51 /* Allocate memory for large vectors */
52 /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional
    array*/
53 #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+
    nbr_of_dimensions*j+k])
54 double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*
    nbr_of_dimensions*sizeof(double));
55
56 double* energy_pot = (double*) malloc(nbr_of_timesteps * sizeof(double)
    );
57 double* energy_kin = (double*) malloc(nbr_of_timesteps * sizeof(double)
    );
58 double* virial = (double*) malloc(nbr_of_timesteps * sizeof(double)
    );
59 double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)
    );
60 double* pressure_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)
    );
61 //double* temperature = (double*) malloc((2 * nbr_of_timesteps_eq +
    nbr_of_timesteps) * sizeof(double));
62 //double* pressure = (double*) malloc((2 * nbr_of_timesteps_eq +
    nbr_of_timesteps) * sizeof(double));
63
64 //TODO go over parameters again
65 /* Initialize parameters*/
66 initial_displacement = 0.05;
67 lattice_param = 4.046; // For aluminium ( )
68 lattice_spacing = lattice_param/sqrt(2.0);
69 timestep = 0.001; // 0.1 Bad, 0.01 Seems decent
70 m_AL = 0.0027964; // In ASU
71 cell_length = 4*lattice_param; // Side of the supercell: The
    256 atoms are
72 // structured in a block of 4
    x4x4 unit cells
73 volume = pow(cell_length, 3);
74
75 // Initialize all displacements, for all times, as 0
76 for (int i = 0; i < nbr_of_timesteps; i++){
77     for (int j = 0; j < nbr_of_particles; j++){
78         for (int k = 0; k < nbr_of_dimensions; k++){
79             qq(i,j,k) = 0;
80         }
81     }
82 }
83
84 /* Put atoms on lattice */
85 init_fcc(q, 4, lattice_param);
86
87
88 /* Initial conditions */
89 for (int i = 0; i < nbr_of_particles; i++){
90     for (int j = 0; j < nbr_of_dimensions; j++){
91
92         // Initial perturbation from equilibrium
93         q[i][j] += lattice_spacing * initial_displacement
94             * ((double)rand()/((double)RAND_MAX);
95     }
96 }
97
98
99
100 get_forces_AL(f, q, cell_length, nbr_of_particles);
101
102 /* Simulation */
103 /* Equilibrium stage */
104
105 double inst_temperature_eq;
106 double inst_pressure_eq;
107 double alpha_T = 1.0;
108 double alpha_P = 1.0;

```



```

109 double energy_kin_eq = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,↵
110 m_AL);
111 double virial_eq = get_virial_AL(q,cell_length,nbr_of_particles);
112 //temperature[0] = instantaneous_temperature(energy_kin_eq, ↵
113     nbr_of_particles);
114 //pressure[0] = instantaneous_pressure(virial_eq, temperature[0], ↵
115     nbr_of_particles, volume);
116
117 for (int equil = 0; equil < 2; equil++) {
118     for (int i = 1; i < nbr_of_timesteps_eq; i++)
119     {
120         /** Verlet algorithm */
121         /* Half step for velocity */
122         for (int j = 0; j < nbr_of_particles; j++){
123             for (int k = 0; k < nbr_of_dimensions; k++){
124                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
125             }
126         }
127
128         /* Update displacement*/
129         for (int j = 0; j < nbr_of_particles; j++){
130             for (int k = 0; k < nbr_of_dimensions; k++){
131                 q[j][k] += timestep * v[j][k];
132             }
133         }
134
135         /* Forces */
136         get_forces_AL(f,q,cell_length,nbr_of_particles);
137
138         /* Final velocity*/
139         for (int j = 0; j < nbr_of_particles; j++){
140             for (int k = 0; k < nbr_of_dimensions; k++){
141                 v[j][k] += timestep * 0.5* f[j][k]/m_AL;
142             }
143         }
144
145         /* Calculate energy */
146         // Kinetic energy
147         energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, ↵
148             nbr_of_particles, m_AL);
149
150         virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);
151
152         inst_temperature_eq = instantaneous_temperature(energy_kin_eq, ↵
153             nbr_of_particles);
154         //temperature[equil*(nbr_of_timesteps_eq-1) + i] = ↵
155             inst_temperature_eq;
156         inst_pressure_eq = instantaneous_pressure(virial_eq, ↵
157             inst_temperature_eq,
158             nbr_of_particles, volume);
159         //pressure[equil*(nbr_of_timesteps_eq-1) + i] = inst_pressure_eq;
160
161         // Update alphas
162         alpha_T = 1.0 + 0.01*(temperature_eq[equil]-inst_temperature_eq)/↵
163             inst_temperature_eq;
164         alpha_P = 1.0 - 0.01*isothermal_compressibility*(pressure_eq - ↵
165             inst_pressure_eq);
166
167         // DEBUG:alpha
168         //printf("%.8f \t %.8f \n", alpha_T, alpha_P);
169
170         // Scale velocities
171         for (int j = 0; j < nbr_of_particles; j++){
172             for (int k = 0; k < nbr_of_dimensions; k++){
173                 v[j][k] *= sqrt(alpha_T);
174             }
175         }
176
177         // Scale positions and volume
178         cell_length *= pow(alpha_P, 1.0/3.0);
179         volume = pow(cell_length, 3);
180         for (int j = 0; j < nbr_of_particles; j++) {
181             for (int k = 0; k < nbr_of_dimensions; k++) {
182                 q[j][k] *= pow(alpha_P, 1.0/3.0);
183             }
184         }
185     }
186 }
187
188 printf("Equilibration done.\n");
189 printf("Cell length: %.8f \n", cell_length);
190
191 for (int i = 0; i < nbr_of_particles; i++){
192     for (int j = 0; j < nbr_of_dimensions; j++){
193         qq(0,i,j)=q[i][j];
194     }
195 }

```

```

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

```

```

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

```

A.5 Task6/MD_main.c

```

1  /*
2  MD_main.c
3
4  Created by Anders Lindman on 2013-10-31.
5  */
6
7  #include <stdio.h>
8  #include <math.h>
9  #include <stdlib.h>
10 #include <time.h>
11 #include "initfcc.h"
12 #include "alpotential.h"
13 #define nbr_of_particles 256
14 #define nbr_of_timesteps 1e4
15 #define nbr_of_timesteps_eq 4000
16 #define nbr_of_dimensions 3
17
18 double boundary_condition(double,double);
19
20
21
22 /* Main program */
23 int main()
24 {
25     srand(time(NULL));
26
27     /* Simulation parameters */
28     double m_AL; // Mass of atom
29     double cell_length; // Side length of supercell
30     double volume;
31     double lattice_spacing; // Smallest length between atoms
32     double initial_displacement; // Initial displacement of the atoms from ←
33         // lattice positions
34     double lattice_param; // Lattice parameter, length of each side in the
35         // unit cell
36     double timestep;
37     double temperature_eq[] = { 1000.0+273.15, 700.0+273.15 };
38     double delta_temperature[] = { -10.0, 10.0 };
39     double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
40
41     FILE *file;
42
43
44     /* Current displacement, velocities, and accelerations */
45     double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
46     double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
47     double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces
48
49     double heat_capacity;
50     double energy_avg[2] = { 0 };
51     double temperature_avg[2] = { 0 };
52
53
54     /* Allocate memory for large vectors */
55
56     double* energy_pot = (double*) malloc(nbr_of_timesteps * sizeof(double)←
57     );
58     double* energy_kin = (double*) malloc(nbr_of_timesteps * sizeof(double)←
59     );
60
61     /* Initialize parameters*/
62     initial_displacement = 0.05;
63     lattice_param = 4.046; // For aluminium ( )
64     lattice_spacing = lattice_param/sqrt(2.0);
65     timestep = 0.001; // 0.1 Bad, 0.01 Seems decent
66     m_AL = 0.0027964; // In ASU
67     cell_length = 4*lattice_param; // Side of the supercell: The ←
68     256 atoms are

```

```

67                                     // structured in a block of 4↔
68                                     x4x4 unit cells
69
70 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             inst_temperature_eq = instantaneous_temperature(energy_kin_eq, ↵
                nbr_of_particles);
140             inst_pressure_eq = instantaneous_pressure(virial_eq, ↵
                inst_temperature_eq,
                nbr_of_particles, volume);
141
142             // Update alphas
143             alpha_T = 1.0 + 0.01*(target_temp-inst_temperature_eq)/↵
                inst_temperature_eq;
144             alpha_P = 1.0 - 0.01*(pressure_eq - inst_pressure_eq);
145
146             // Scale velocities
147             for (int j = 0; j < nbr_of_particles; j++){
148                 for (int k = 0; k < nbr_of_dimensions; k++){
149

```

```

152         v[j][k] *= sqrt(alpha_T);
153     }
154 }
155
156 // Scale positions and volume
157 cell_length *= pow(alpha_P, 1.0/3.0);
158 volume = pow(cell_length, 3);
159 for (int j = 0; j < nbr_of_particles; j++) {
160     for (int k = 0; k < nbr_of_dimensions; k++) {
161         q[j][k] *= pow(alpha_P, 1.0/3.0);
162     }
163 }
164
165 }
166 }
167
168 // Compute energies, temperature etc. at equilibrium
169 energy_pot[0] = get_energy_AL(q, cell_length, nbr_of_particles);
170 energy_kin[0] = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, ←
171     m_AL);
172
173
174 /* Simulation after equilibrium*/
175 for (int i = 1; i < nbr_of_timesteps; i++)
176 {
177     /** Verlet algorithm **/
178     /* Half step for velocity */
179     for (int j = 0; j < nbr_of_particles; j++){
180         for (int k = 0; k < nbr_of_dimensions; k++){
181             v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
182         }
183     }
184
185     /* Update displacement*/
186     for (int j = 0; j < nbr_of_particles; j++){
187         for (int k = 0; k < nbr_of_dimensions; k++){
188             q[j][k] += timestep * v[j][k];
189         }
190     }
191
192     /* Update Forces */
193     get_forces_AL(f, q, cell_length, nbr_of_particles);
194
195     /* Final 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     /* Calculate energy */
203     // Potential energy
204     energy_pot[i] = get_energy_AL(q, cell_length, nbr_of_particles);
205     // Kinetic energy
206     energy_kin[i] = get_kinetic_AL(v, nbr_of_dimensions, ←
207         nbr_of_particles, m_AL);
208
209     } // equilibration/simulation
210
211     // Compute heat capacity
212     temperature_avg[d] = averaged_temperature(energy_kin, nbr_of_particles, ←
213         nbr_of_timesteps-1);
214     // Compute average total energy
215     for (int i = 0; i < nbr_of_timesteps; i++)
216         energy_avg[d] += energy_pot[i] + energy_kin[i];
217     energy_avg[d] /= nbr_of_timesteps;
218
219     printf("Temp: %f\nAverage total energy: %.10f\n", temperature_avg[d], ←
220         energy_avg[d]);
221 }
222
223 // Compute heat capacity
224 heat_capacity = (energy_avg[1]-energy_avg[0])/(temperature_avg[1]-←
225     temperature_avg[0]);
226
227 printf("heat capacity: %f\n", heat_capacity);
228
229 // Save results to file
230 file = fopen("heat_capacity.dat", "w");
231 fprintf(file, "%.2f\t\t\t\t\t", temperature_eq[1], heat_capacity);
232 fclose(file);
233
234 free(energy_kin);        energy_kin = NULL;
235 free(energy_pot);        energy_pot = NULL;
236
237 return 0;

```

A.6 Task7/MD_main.c

```

1  /*
2  MD_main.c
3
4  Created by Anders Lindman on 2013-10-31.
5  */
6
7  #include <stdio.h>
8  #include <math.h>
9  #include <stdlib.h>
10 #include <time.h>
11 #include "initfcc.h"
12 #include "alpotential.h"
13 #define nbr_of_particles 256
14 #define nbr_of_timesteps 1000
15 #define nbr_of_timesteps_eq 4000
16 #define nbr_of_dimensions 3
17
18 #define PI 3.141592653589
19 int get_bin(double , double , double , double );
20
21 double boundary_condition_dist_sq(double u1[3], double u2[3], double L);
22
23 /* Main program */
24 int main()
25 {
26     srand(time(NULL));
27
28     /* Simulation parameters */
29     double m_AL; // Mass of atom
30     double cell_length; // Side length of supercell
31     double volume;
32     double lattice_spacing; // Smallest length between atoms
33     double initial_displacement; // Initial displacement of the atoms from ←
34         their // lattice positions
35     double lattice_param; // Lattice parameter, length of each side in the
36         // unit cell
37     double timestep;
38     double temperature_eq[] = { 1500.0+273.15, 700.0+273.15 };
39     double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
40     double isothermal_compressibility = 1.0; //0.8645443196; // 1.385e-11 m^2/N ←
41         = 1.385/1.602 ^3/eV
42
43     FILE *file;
44
45     /* Current displacement, velocities, and acceleratons */
46     double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
47     double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
48     double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces
49
50     /* Allocate memory for large vectors */
51     /* Simulate 3 dimensional data by placing initializeing a 1-dimensional ←
52         array*/
53     #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+←
54         nbr_of_dimensions*j+k])
55     double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*←
56         nbr_of_dimensions*sizeof(double));
57
58     double* energy = (double*) malloc(nbr_of_timesteps * sizeof(double)←
59         );
60     double* energy_kin = (double*) malloc(nbr_of_timesteps * sizeof(double)←
61         );
62     double* virial = (double*) malloc(nbr_of_timesteps * sizeof(double)←
63         );
64     double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)←
65         );
66     double* pressure_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)←
67         );
68     double* temperature = (double*) malloc((2 * nbr_of_timesteps_eq + ←
69         nbr_of_timesteps) * sizeof(double));
70     double* pressure = (double*) malloc((2 * nbr_of_timesteps_eq + ←
71         nbr_of_timesteps) * sizeof(double));
72
73     int k_bins = 250;
74
75     //TODO go over parameters again
76     /* Initialize parameters*/
77     initial_displacement = 0.05;
78     lattice_param = 4.046; // For aluminium ( )
79     lattice_spacing = lattice_param/sqrt(2.0);

```

```

71 timestep                = 0.01; // 0.1 Bad, 0.01 Seems decent
72 m_AL                    = 0.0027964; // In ASU
73 cell_length             = 4*lattice_param; // Side of the supercell: The ←
    256 atoms are
74                                     // structured in a block of 4←
                                     x4x4 unit cells
75 volume                  = pow(cell_length, 3);
76
77 // Initialize all displacements, for all times, as 0
78 for (int i = 0; i < nbr_of_timesteps; i++) {
79     for (int j = 0; j < nbr_of_particles; j++) {
80         for (int k = 0; k < nbr_of_dimensions; k++) {
81             qq(i,j,k) = 0;
82         }
83     }
84 }
85
86 /* Put atoms on lattice */
87 init_fcc(q, 4, lattice_param);
88
89 /* Initial conditions */
90 for (int i = 0; i < nbr_of_particles; i++) {
91     for (int j = 0; j < nbr_of_dimensions; j++) {
92
93         // Initial perturbation from equilibrium
94         q[i][j] += lattice_spacing * initial_displacement
95             * ((double)rand()/((double)RAND_MAX));
96     }
97 }
98
99
100
101
102 get_forces_AL(f, q, cell_length, nbr_of_particles);
103
104 /* Simulation */
105 /* Equilibrium stage */
106
107 double inst_temperature_eq;
108 double inst_pressure_eq;
109 double alpha_T = 1.0;
110 double alpha_P = 1.0;
111 double energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, ←
    m_AL);
112 double virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);
113
114 temperature[0] = instantaneous_temperature(energy_kin_eq, nbr_of_particles)←
    ;
115 pressure[0] = instantaneous_pressure(virial_eq, temperature[0], ←
    nbr_of_particles, volume);
116
117 for (int equil = 0; equil < 2; equil++) {
118     for (int i = 1; i < nbr_of_timesteps_eq; i++) {
119
120         /** Verlet algorithm */
121         /* Half step for velocity */
122         for (int j = 0; j < nbr_of_particles; j++) {
123             for (int k = 0; k < nbr_of_dimensions; k++) {
124                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
125             }
126         }
127
128         /* Update displacement*/
129         for (int j = 0; j < nbr_of_particles; j++) {
130             for (int k = 0; k < nbr_of_dimensions; k++) {
131                 q[j][k] += timestep * v[j][k];
132             }
133         }
134
135         /* Forces */
136         get_forces_AL(f, q, cell_length, nbr_of_particles);
137
138         /* Final velocity*/
139         for (int j = 0; j < nbr_of_particles; j++) {
140             for (int k = 0; k < nbr_of_dimensions; k++) {
141                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
142             }
143         }
144
145         /* Calculate energy */
146         // Kinetic energy
147         energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, ←
            nbr_of_particles, m_AL);
148
149         virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);
150
151
152         inst_temperature_eq = instantaneous_temperature(energy_kin_eq, ←
            nbr_of_particles);
153         temperature[equil*(nbr_of_timesteps_eq-1) + i] = inst_temperature_eq←
            ;

```

```

154     inst_pressure_eq = instantaneous_pressure(virial_eq, ←
155         inst_temperature_eq,
156         nbr_of_particles, volume);
157     pressure[equil*(nbr_of_timesteps_eq-1) + i] = inst_pressure_eq;
158
159     // Update alphas
160     alpha_T = 1.0 + 0.01*(temperature_eq[equil]-inst_temperature_eq)/←
161         inst_temperature_eq;
162     alpha_P = 1.0 - 0.01*isothermal_compressibility*(pressure_eq - ←
163         inst_pressure_eq);
164
165     // Scale velocities
166     for (int j = 0; j < nbr_of_particles; j++) {
167         for (int k = 0; k < nbr_of_dimensions; k++) {
168             v[j][k] *= sqrt(alpha_T);
169         }
170     }
171
172     // Scale positions and volume
173     cell_length *= pow(alpha_P, 1.0/3.0);
174     volume = pow(cell_length, 3);
175     for (int j = 0; j < nbr_of_particles; j++) {
176         for (int k = 0; k < nbr_of_dimensions; k++) {
177             q[j][k] *= pow(alpha_P, 1.0/3.0);
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 // Compute energies, temperature etc. at equilibrium
190 double min = 0.0;
191 double max = sqrt(3*cell_length*cell_length);
192 double d_r = (max-min)/(1.0*k_bins);
193 int bins[k_bins];
194 int* bins2 = (int*) malloc(k_bins * sizeof(int));
195
196 for (int i = 0; i < k_bins; i++) {
197     bins[i]=0;
198     bins2[i]=0;
199 }
200
201 for (int i = 1; i < nbr_of_timesteps; i++)
202 {
203     /** Verlet algorithm **/
204     /* Half step for velocity */
205     for (int j = 0; j < nbr_of_particles; j++){
206         for (int k = 0; k < nbr_of_dimensions; k++){
207             v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
208         }
209     }
210
211     /* Update displacement*/
212     for (int j = 0; j < nbr_of_particles; j++){
213         for (int k = 0; k < nbr_of_dimensions; k++){
214             q[j][k] += timestep * v[j][k];
215         }
216     }
217
218     /* Forces */
219     get_forces_AL(f,q,cell_length,nbr_of_particles);
220
221     /* Final velocity*/
222     for (int j = 0; j < nbr_of_particles; j++){
223         for (int k = 0; k < nbr_of_dimensions; k++){
224             v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
225         }
226     }
227
228     /* Calculate energy */
229     // Potential energy
230     energy[i] = get_energy_AL(q,cell_length,nbr_of_particles);
231     // Kinetic energy
232     energy_kin[i] = get_kinetic_AL(v,nbr_of_dimensions,nbr_of_particles,m_AL←
233 );
234
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 // Create Histogram
246
247 for (int i = 1; i < nbr_of_timesteps; i++)
248 {
249     for (int j = 1 ; j < nbr_of_particles; j++) {
250         for (int k = j+1 ; k < nbr_of_particles; k++) {
251
252             double q1[nbr_of_dimensions];
253             double q2[nbr_of_dimensions];
254             for (int d = 0; d < nbr_of_dimensions; d++) {
255                 q1[d] = qq(i,j,d);
256                 q2[d] = qq(i,k,d);
257             }
258             double distance_sq = boundary_condition_dist_sq(q1, q2, ←
259                 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], ←
279         Nideal[i]);
280 }
281 fclose(file);
282 // TO THIS ISH TODO
283
284 free(energy_kin);      energy_kin = NULL;
285 free(energy);          energy = NULL;
286 free(displacement);    displacement = 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

```
1  /*
2  MD_main.c
3
4  Created by Anders Lindman on 2013-10-31.
5  */
6
7  #include <stdio.h>
8  #include <math.h>
9  #include <stdlib.h>
10 #include <time.h>
11 #include "initfcc.h"
12 #include "alpotential.h"
13 #include <complex.h>
14 #define nbr_of_particles 256
15 #define nbr_of_timesteps 1000
16 #define nbr_of_timesteps_eq 4000
17 #define nbr_of_dimensions 3
18
19 #define PI 3.141592653589
20 int get_bin(double , double , double , double );
21
22 double boundary_condition(double,double);
23
24 /* Main program */
25 int main()
26 {
27     srand(time(NULL));
28
29     /* Simulation parameters */
30     double m_AL; // Mass of atom
31     double cell_length; // Side length of supercell
32     double volume;
33     double lattice_spacing; // Smallest length between atoms
34     double initial_displacement; // Initial displacement of the atoms from ←
35         // lattice positions
36     double lattice_param; // Lattice parameter, length of each side in the
37         // unit cell
38     double timestep;
39     double temperature_eq[] = { 1500.0+273.15, 700.0+273.15 };
40     double pressure_eq = 101325e-11/1.602; // 1 atm in ASU
41     double isothermal_compressibility = 1.0; //0.8645443196; // 1.385e-11 m^2/N ←
42         // = 1.385/1.602 ^3/eV
43
44     FILE *file;
45
46     /* Current displacement, velocities, and acceleratons */
47     double q[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Displacements
48     double v[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Velocities
49     double f[nbr_of_particles][nbr_of_dimensions] = { 0 }; // Forces
50
51     /* Allocate memory for large vectors */
52     /* Simulate 3 dimensional data by placing iniitalizeing a 1-dimensional ←
53         array*/
54     #define qq(i,j,k) (disp_arr[nbr_of_particles*nbr_of_dimensions*i+←
55         nbr_of_dimensions*j+k])
56     double* disp_arr = (double*)malloc(nbr_of_timesteps*nbr_of_particles*←
57         nbr_of_dimensions*sizeof(double));
58
59     double* energy = (double*) malloc(nbr_of_timesteps * sizeof(double)←
60         );
61     double* energy_kin = (double*) malloc(nbr_of_timesteps * sizeof(double)←
62         );
63     double* virial = (double*) malloc(nbr_of_timesteps * sizeof(double)←
64         );
65     double* temperature_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)←
66         );
67     double* pressure_avg = (double*) malloc(nbr_of_timesteps * sizeof(double)←
68         );
69     double* temperature = (double*) malloc((2 * nbr_of_timesteps_eq + ←
70         nbr_of_timesteps) * sizeof(double));
71     double* pressure = (double*) malloc((2 * nbr_of_timesteps_eq + ←
72         nbr_of_timesteps) * sizeof(double));
73
74     //TODO go over parameters again
75     /* Initialize parameters*/
76     initial_displacement = 0.05;
77     lattice_param = 4.046; // For aluminium ( )
78     lattice_spacing = lattice_param/sqrt(2.0);
79     timestep = 0.01; // 0.1 Bad, 0.01 Seems decent
80     m_AL = 0.0027964; // In ASU
81     cell_length = 4*lattice_param; // Side of the supercell: The ←
82         256 atoms are
```

```

74                                     // structured in a block of 4↔
75                                     x4x4 unit cells
76
77 volume                             = pow(cell_length, 3);
78
79 // Initialize all displacements, for all times, as 0
80 for (int i = 0; i < nbr_of_timesteps; i++){
81     for (int j = 0; j < nbr_of_particles; j++){
82         for (int k = 0; k < nbr_of_dimensions; k++){
83             qq(i,j,k) = 0;
84         }
85     }
86 }
87
88 /* Put atoms on lattice */
89 init_fcc(q, 4, lattice_param);
90
91 /* Initial conditions */
92 for (int i = 0; i < nbr_of_particles; i++){
93     for (int j = 0; j < nbr_of_dimensions; j++){
94         // Initial perturbation from equilibrium
95         q[i][j] += lattice_spacing * initial_displacement
96             * ((double)rand()/(double)RAND_MAX);
97     }
98 }
99
100
101
102 get_forces_AL(f, q, cell_length, nbr_of_particles);
103
104 /* Simulation */
105 /* Equilibrium stage */
106
107 double inst_temperature_eq;
108 double inst_pressure_eq;
109 double alpha_T = 1.0;
110 double alpha_P = 1.0;
111 double energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, nbr_of_particles, ↵
112     m_AL);
113 double virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);
114
115 temperature[0] = instantaneous_temperature(energy_kin_eq, nbr_of_particles)↵
116 ;
117 pressure[0] = instantaneous_pressure(virial_eq, temperature[0], ↵
118     nbr_of_particles, volume);
119
120 for (int equil = 0; equil < 2; equil++) {
121     for (int i = 1; i < nbr_of_timesteps_eq; i++)
122     {
123         /** Verlet algorithm */
124         /* Half step for velocity */
125         for (int j = 0; j < nbr_of_particles; j++){
126             for (int k = 0; k < nbr_of_dimensions; k++){
127                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
128             }
129         }
130
131         /* Update displacement*/
132         for (int j = 0; j < nbr_of_particles; j++){
133             for (int k = 0; k < nbr_of_dimensions; k++){
134                 q[j][k] += timestep * v[j][k];
135             }
136         }
137
138         /* Forces */
139         get_forces_AL(f, q, cell_length, nbr_of_particles);
140
141         /* Final velocity*/
142         for (int j = 0; j < nbr_of_particles; j++){
143             for (int k = 0; k < nbr_of_dimensions; k++){
144                 v[j][k] += timestep * 0.5 * f[j][k]/m_AL;
145             }
146         }
147
148         /* Calculate energy */
149         // Kinetic energy
150         energy_kin_eq = get_kinetic_AL(v, nbr_of_dimensions, ↵
151             nbr_of_particles, m_AL);
152
153         virial_eq = get_virial_AL(q, cell_length, nbr_of_particles);
154
155         inst_temperature_eq = instantaneous_temperature(energy_kin_eq, ↵
156             nbr_of_particles);
157         temperature[equil*(nbr_of_timesteps_eq-1) + i] = inst_temperature_eq↵
158 ;
159         inst_pressure_eq = instantaneous_pressure(virial_eq, ↵
160             inst_temperature_eq,
161             nbr_of_particles, volume);
162         pressure[equil*(nbr_of_timesteps_eq-1) + i] = inst_pressure_eq;

```

```

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

```

```

241 // Pressure
242 pressure_avg[i] = averaged_pressure(virial, energy_kin, volume, i);
243 pressure[2*(nbr_of_timesteps_eq-1) + i] = instantaneous_pressure(virial[←
    i],
244     temperature[2*(nbr_of_timesteps_eq-1) + i],
245     nbr_of_particles, volume);
246
247 /* Save current displacements to array*/
248 for (int j = 0; j < nbr_of_particles; j++){
249     for (int k = 0; k < nbr_of_dimensions; k++){
250         qq(i,j,k)=q[j][k];
251     }
252 }
253 }
254 } // equilibration/simulation
255
256 int n_x = 30;
257 int n_y = 30;
258 int n_z = 30;
259
260 double factor = PI*2.0/cell_length;
261
262 double qS[n_x][n_y][n_z][3];
263 for (int i = 0; i < n_x; i++)
264     for (int j = 0; j < n_y ; j++)
265         for (int k = 0; k < n_z; k++){
266             qS[i][j][k][0]=i*factor;
267             qS[i][j][k][1]=j*factor;
268             qS[i][j][k][2]=k*factor;
269         }
270
271 double s[n_x][n_y][n_z];
272 for (int i = 0; i < n_x; i++)
273     for (int j = 0; j < n_y ; j++)
274         for (int k = 0; k < n_z; k++)
275             {
276                 if ( !((i==j) && (i==k) && (i==0))) {
277                     double complex sum = 0;
278                     for (int r=0; r < nbr_of_particles; r++)
279                         {
280                             double complex expo=0;
281                             for (int d = 0; d < nbr_of_dimensions; d++)
282                                 {
283                                     double ri = q[r][d];
284                                     ri=boundary_condition(ri,cell_length);
285                                     expo+= qS[i][j][k][d]*ri;
286                                 }
287                             expo=expo*I;
288                             sum+= cexp(expo);
289                         }
290                     sum = cabs(sum);
291                     sum=sum*sum/nbr_of_particles;
292                     s[i][j][k]=sum;
293                 }
294             }
295
296 double data[n_x*n_y*n_z];
297 double dis[n_x*n_y*n_z];
298 int iterator =0;
299 for (int i = 0; i < n_x; i++)
300     for (int j = 0; j < n_y ; j++)
301         for (int k = 0; k < n_z; k++)
302             {
303                 dis[iterator] =sqrt(1.0*i*i+1.0*j*j+1.0*k*k);
304                 data[iterator] = s[i][j][k];
305                 iterator++;
306             }
307 double max =0;
308 double min = 1e10;
309 for (int i = 0; i < n_x*n_y*n_z; i++ )
310     {
311         if (dis[i] > max)
312             max = dis[i];
313         if (dis[i] < min)
314             min = dis[i];
315     }
316
317 int k_bins=200;
318 double d_r = (max-min)/(1.0*k_bins);
319 int bins[k_bins];
320 for (int i = 0; i < n_x*n_y*n_z; i++)
321     {
322         int bin = get_bin(data[i],min,max,d_r);
323         bins[bin]++;
324     }
325
326 file = fopen("data.dat","w");
327 for (int i = 0; i < k_bins; i++)
328     {
329         fprintf(file, "%e \t %i \n", (double)(min+d_r*i*1.0), bins[i]);
330     }

```

```

331     fclose(file);
332     /*
333     file = fopen("data.dat","w");
334     for (int i = 0; i < n_x*n_y*n_z; i++)
335     {
336         fprintf(file, "%e \t %e \n",dis[i],data[i] );
337     }
338     fclose(file);*/
339
340     free(energy_kin);      energy_kin = NULL;
341     free(energy);          energy = NULL;
342     free(dis_arry);        dis_arry = NULL;
343     free(virial);          virial = NULL;
344     free(temperature_avg); temperature_avg = NULL;
345     free(pressure_avg);    pressure_avg = NULL;
346
347     return 0;
348 }
349
350
351 int get_bin(double val , double min , double max , double d_r)
352 {
353     int bin =0;
354     double current=min;
355     while (current <= val)
356     {
357         current += d_r;
358         bin++;
359     }
360     return bin;
361 }
362
363 double boundary_condition(double u, double L)
364 {
365
366     double f = fmod(u,L);
367     if (f < 0)
368         return -f;
369     else
370         return f;
371 }

```

```

1  /*
2  alpotential.c
3  Program that contains functions that calculate properties (potential energy, ←
4  forces, etc.) of a set of Aluminum atoms using an embedded atom model (EAM←
5  ) potential.
6  Created by Anders Lindman on 2013-03-14.
7  */
8
9  #include <stdio.h>
10 #include <math.h>
11 #include <stdlib.h>
12
13 /*Parameters for the AL EAM potential */
14 #define PAIR_POTENTIAL_ROWS 18
15 const double pair_potential[90] = {2.0210, 2.2730, 2.4953, 2.7177, 2.9400, ←
16 3.1623, 3.3847, 3.6070, 3.8293, 4.0517, 4.2740, 4.4963, 4.7187, 4.9410, ←
17 5.1633, 5.3857, 5.6080, 6.0630, 2.0051, 0.7093, 0.2127, 0.0202, -0.0386, ←
18 -0.0492, -0.0424, -0.0367, -0.0399, -0.0574, -0.0687, -0.0624, -0.0492, ←
19 -0.0311, -0.0153, -0.0024, -0.0002, 0, -7.2241, -3.3383, -1.3713, -0.4753, ←
20 -0.1171, 0.0069, 0.0374, 0.0122, -0.0524, -0.0818, -0.0090, 0.0499, 0.0735, ←
21 0.0788, 0.0686, 0.0339, -0.0012, 0, 9.3666, 6.0533, 2.7940, 1.2357, ←
22 0.3757, 0.1818, -0.0445, -0.0690, -0.2217, 0.0895, 0.2381, 0.0266, 0.0797, ←
23 -0.0557, 0.0097, -0.1660, 0.0083, 0, -4.3827, -4.8865, -2.3363, -1.2893, ←
24 -0.2907, -0.3393, -0.0367, -0.2290, 0.4667, 0.2227, -0.3170, 0.0796, ←
25 -0.2031, 0.0980, -0.2634, 0.2612, -0.0102, 0};
26
27 #define ELECTRON_DENSITY_ROWS 15
28 const double electron_density[75] = {2.0210, 2.2730, 2.5055, 2.7380, 2.9705, ←
29 3.2030, 3.4355, 3.6680, 3.9005, 4.1330, 4.3655, 4.5980, 4.8305, 5.0630, ←
30 6.0630, 0.0824, 0.0918, 0.0883, 0.0775, 0.0647, 0.0512, 0.0392, 0.0291, ←
31 0.0186, 0.0082, 0.0044, 0.0034, 0.0027, 0.0025, 0.0000, 0.0707, 0.0071, ←
32 -0.0344, -0.0533, -0.0578, -0.0560, -0.0465, -0.0428, -0.0486, -0.0318, ←
33 -0.0069, -0.0035, -0.0016, -0.0008, 0, -0.1471, -0.1053, -0.0732, -0.0081, ←
34 -0.0112, 0.0189, 0.0217, -0.0056, -0.0194, 0.0917, 0.0157, -0.0012, 0.0093, ←
35 -0.0059, 0, 0.0554, 0.0460, 0.0932, -0.0044, 0.0432, 0.0040, -0.0392, ←
36 -0.0198, 0.1593, -0.1089, -0.0242, 0.0150, -0.0218, 0.0042, 0};
37
38 #define EMBEDDING_ENERGY_ROWS 13
39 const double embedding_energy[65] = {0, 0.1000, 0.2000, 0.3000, 0.4000, 0.5000, ←
40 0.6000, 0.7000, 0.8000, 0.9000, 1.0000, 1.1000, 1.2000, 0, -1.1199, ←
41 -1.4075, -1.7100, -1.9871, -2.2318, -2.4038, -2.5538, -2.6224, -2.6570, ←
42 -2.6696, -2.6589, -2.6358, -18.4387, -5.3706, -2.3045, -3.1161, -2.6175, ←
43 -2.0666, -1.6167, -1.1280, -0.4304, -0.2464, -0.0001, 0.1898, 0.2557, ←
44 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};

22
23
24 #define k_b 0.00008617 // (eV)
25
26 /* Evaluates the spline in x. */
27
28 double splineEval(double x, const double *table, int m) {
29     /* int m = mxGetM(spline), i, k; */
30     int i, k;
31
32     /*double *table = mxGetPr(spline);*/
33     double result;
34
35     int k_lo = 0, k_hi = m;
36
37     /* Find the index by bisection. */
38     while (k_hi - k_lo > 1) {
39         k = (k_hi + k_lo) >> 1;
40         if (table[k] > x)
41             k_hi = k;
42         else
43             k_lo = k;
44     }
45
46     /* Switch to local coord. */
47     x -= table[k_lo];
48
49     /* Horner's scheme */
50     result = table[k_lo + 4*m];
51     for (i = 3; i > 0; i--) {
52         result *= x;
53         result += table[k_lo + i*m];
54     }
55
56     return result;
57 }
58
59 /* Evaluates the derivative of the spline in x. */
60
61 double splineEvalDiff(double x, const double *table, int m) {
62     /*int m = mxGetM(spline), i, k;
63     double *table = mxGetPr(spline);
64     */
65     int i, k;
66     double result;
67
68     int k_lo = 0, k_hi = m;
69
70     /* Find the index by bisection. */
71     while (k_hi - k_lo > 1) {
72         k = (k_hi + k_lo) >> 1;
73         if (table[k] > x)
74             k_hi = k;
75         else
76             k_lo = k;
77     }
78
79     /* Switch to local coord. */
80     x -= table[k_lo];
81
82     /* Horner's scheme */
83     result = 3*table[k_lo + 4*m];
84     for (i = 3; i > 1; i--) {
85         result *= x;
86         result += (i-1)*table[k_lo + i*m];
87     }
88
89     return result;
90 }
91
92 /* Returns the forces */
93 void get_forces_AL(double forces[][3], double positions[][3], double cell_length↵
, int nbr_atoms)
94 {
95     int i, j;
96     double cell_length_inv, cell_length_sq;
97     double rcut, rcut_sq;
98     double densityi, dens, drho_dr, force;
99     double dUpair_dr;
100     double sxi, syi, szi, sxij, syij, szij, rij, rij_sq;
101
102     double *sx = malloc(nbr_atoms * sizeof (double));
103     double *sy = malloc(nbr_atoms * sizeof (double));
104     double *sz = malloc(nbr_atoms * sizeof (double));
105     double *fx = malloc(nbr_atoms * sizeof (double));
106     double *fy = malloc(nbr_atoms * sizeof (double));
107     double *fz = malloc(nbr_atoms * sizeof (double));
108

```

```

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

```



```

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

```

```

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

```

```

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 */
371         if (rij_sq < rcut_sq) {
372             rij = sqrt(rij_sq);
373             dens = splineEval(rij, electron_density, ELECTRON_DENSITY_ROWS);
374             densityi += dens;
375             density[j] += dens;
376         }
377     }
378     density[i] = densityi;
379 }
380
381 /* Loop over atoms to calculate derivative of embedding function
382 and embedding function. */
383 for (i = 0; i < nbr_atoms; i++) {
384     dUembed_drho[i] = splineEvalDiff(density[i], embedding_energy, ←
        EMBEDDING_ENERGY_ROWS);
385 }
386
387 /* Compute forces on atoms. */
388 /* Loop over atoms again :-(. */
389
390 virial = 0;
391
392 for (i = 0; i < nbr_atoms; i++) {
393     /* Periodically translate coords of current particle to positive quadrants ←
        */
394     sxi = sx[i] - floor(sx[i]);
395     syi = sy[i] - floor(sy[i]);
396     szi = sz[i] - floor(sz[i]);
397
398     densityi = density[i];
399
400     /* Loop over other atoms. */
401     for (j = i + 1; j < nbr_atoms; j++) {
402         /* Periodically translate atom j to positive quadrants and calculate ←
            distance to it. */
403         sxij = sxi - (sx[j] - floor(sx[j]));
404         syij = syi - (sy[j] - floor(sy[j]));
405         szij = szi - (sz[j] - floor(sz[j]));
406
407         /* Periodic boundary conditions. */
408         sxij = sxij - (int)floor(sxij + 0.5);
409         syij = syij - (int)floor(syij + 0.5);
410         szij = szij - (int)floor(szij + 0.5);
411
412         /* squared distance between atom i and j */
413         rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
414
415         /* Add force and energy contribution if distance between atoms smaller ←
            than rcut */
416         if (rij_sq < rcut_sq) {
417             rij = sqrt(rij_sq);
418             dUpair_dr = splineEvalDiff(rij, pair_potential, PAIR_POTENTIAL_ROWS);
419             drho_dr = splineEvalDiff(rij, electron_density, ←
                ELECTRON_DENSITY_ROWS);
420
421             /* Add virial contribution from i-j interaction */
422             force = -(dUpair_dr + (dUembed_drho[i] + dUembed_drho[j])*←
                drho_dr) / rij;
423
424             virial += force * rij_sq;
425         }
426     }
427 }
428
429 virial /= 3.0;
430
431 free(sx); free(sy); free(sz); sx = NULL; sy = NULL; sz = NULL;
432 free(density); density = NULL;
433 free(dUembed_drho); dUembed_drho = NULL;
434
435 return(virial);
436 }
437
438 double get_kinetic_AL(double velocities[][3], int nbr_of_dimensions, int ←
    nbr_atoms, double m_AL)
439 {
440     double energy = 0;
441     for (int j = 0; j < nbr_atoms; j++) {
442         for (int k = 0; k < nbr_of_dimensions; k++) {
443             energy += m_AL * pow(velocities[j][k], 2) / 2.0;
444         }
445     }
446     return energy;
447 }
448
449
450

```

```

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

```