

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

H1b: Molecular dynamics simulation

Michael Högberg & Simon Jacobsson

December 4, 2019

Task N°	Points	Avail. points
Σ		

Introduction

In this report we will present a numerical method to solve for molecular dynamics. First we present some basic methods for setting up a system to study this will be done in the settings of 256 aluminium atoms in a face-centered cubic (fcc) structure with periodic boundary conditions, i.e. an infinite system.

Task 1

The Lennard-Jones potential for aluminium in an fcc structure is implemented as is shown in Appendix A.4 given in the code file `alpotential.c`. The potential depends on the spacing between the atoms which is given by the structure, fcc, and the lattice constant. In Figure 1 we can see the potential energy as a function of the unit cell-volume, that is we find the lattice constant at 0 K. The lattice constant is thus $\sqrt[3]{65.5} \approx 4.03$.

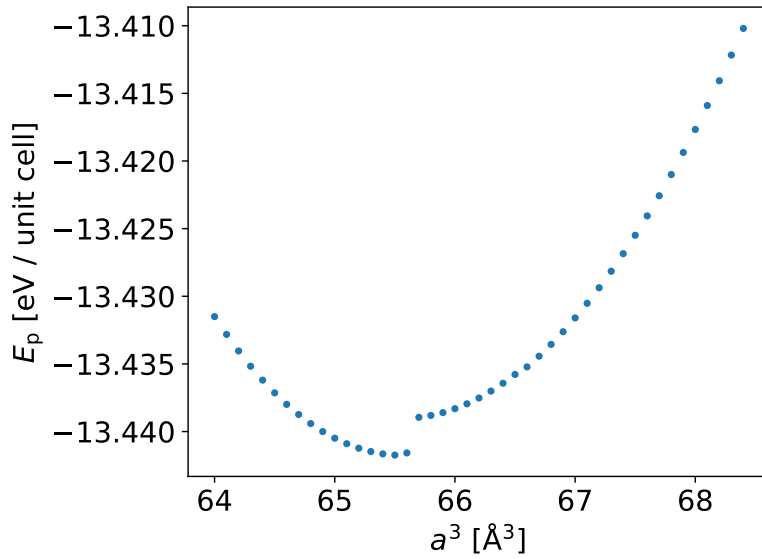


Figure 1: Shows potential energy as a function of unit-cell volume for aluminium. That is the minimum gives the lattice constant at 0 K, $\sqrt[3]{65.5} \approx 4.03$

Task 2

To be able to investigate the system at non-zero temperature we need to introduce some energy, we do this by displacing all atoms from their origins. The displacement is done randomly for each particle with uniform distribution $\pm 6.5\%$ from their origin, the randomness is due to not introduce a net velocity of the whole system, which wouldn't contribute to the temperature.

The time evolution of the system is done with the Verlet algorithm,

$$\begin{cases} \mathbf{v}(t + \frac{\Delta t}{2}) = \mathbf{v}(t) + \frac{1}{2}\mathbf{a}(t)\Delta t \\ \mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t + \frac{\Delta t}{2})\Delta t \\ \text{Update accelerations by } \text{get_forces_AL}() \\ \mathbf{v}(t + \Delta t) = \mathbf{v}(t + \frac{\Delta t}{2}) + \frac{1}{2}\mathbf{a}(t + \Delta t)\Delta t \end{cases} \quad (1)$$

where `get_forces_AL()` is a method provided in `alpotential.c`, see Appendix A.4. This is how the velocities arise from the displaced initial state.

In Figure 2, the system is time evolved using a small time step $\Delta t = 0.001$ ps. We can see that the system has a temperature of around 800 K, this is not exact since we initiate the system randomly and the total energy initiate is roughly the mean displacement and

256 atoms is still a finite and not a too big number. As stated, the system is initiated purely by offsetting the particles from their stationary positions, and thus the total energy is purely potential at $t = 0$. Notice also that the kinetic energy starts out at 0 eV. It seems it takes around 0.2 ps for the system to "forget" this initialization. We call this *equilibration*.

Since our system is closed, the total energy should be constant, while the potential and kinetic energy will have some energy exchange even after the system is equilibrated. We can see that for sufficiently small time step, 0.001 ps in Figure 2, we do have energy conservation but when increasing the time step to 0.01 ps, Figure 3, the total energy oscillates a bit around its average. Increasing the time step further, to ~ 0.021 ps in Figure 4, the energy conservation is totally lost. The conclusion thus is that we should use a time step no larger than 0.01 ps. We will in the following tasks use a time step on the order of 0.001 ps.

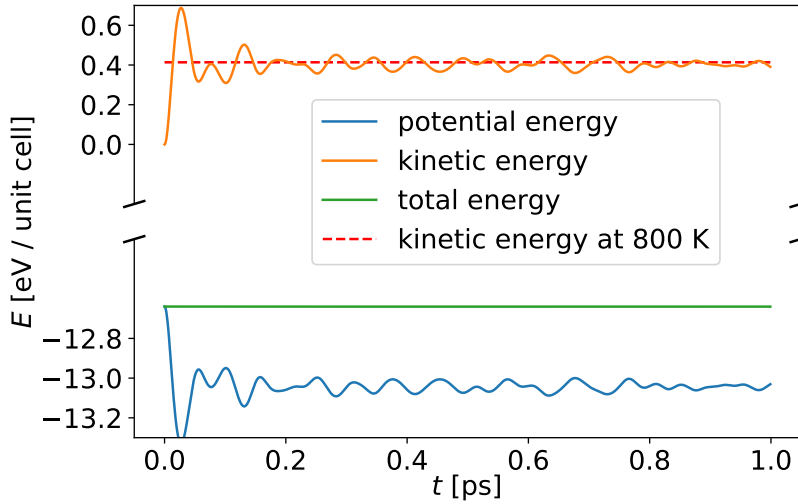


Figure 2: The time evolution of the kinetic, potential, and total energy per unit cell, calculated with time step 0.001 ps. After about 0.2 ps, the oscillations between kinetic and potential energy appear to have stabilized. The total energy is also clearly conserved throughout the process.

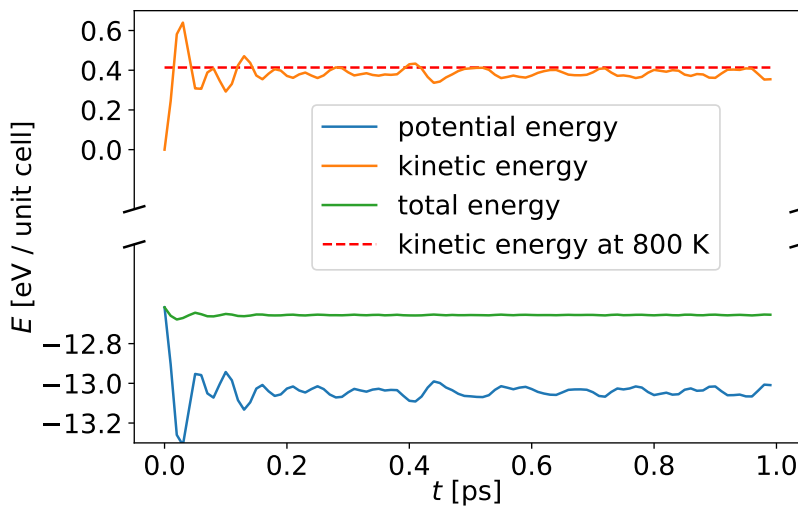


Figure 3: The kinetic, potential, and total energy per unit cell, calculated with time step 0.01 ps. As in Figure 2, there appear to be larger oscillations between kinetic and potential energy within 0.2 ps of start. Unlike in Figure 2, the total energy seems to oscillate around its average value.

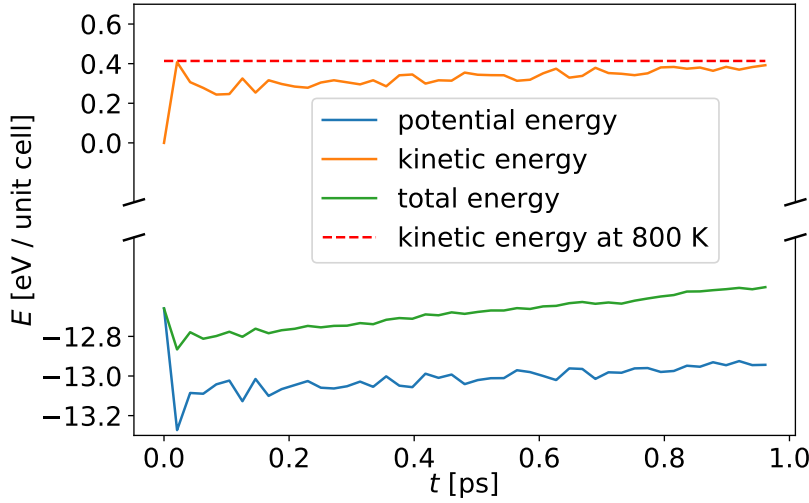


Figure 4: The kinetic, potential, and total energy per unit cell, calculated with time step ~ 0.021 ps. The total energy is clearly not conserved throughout the process, implying that this is a too large time step.

Task 3

When initiating a system, the parameters of interest is rarely that of displaced atoms together with their initial velocity, but that of temperature and pressure. There is no analytic relation of how to go from a specific temperature and pressure to displaced atoms and velocities, the procedure will therefore be that of iteration. That is first starting with a system and calculate the temperature and pressure and checking that to a desired values and then change the system and calculate new temperature and pressure and going on for some time until the desired values are reached.

Before going into equilibration, we initiate the system as in Task 2, i.e. a random displacement of the atoms. Which is followed by running the Verlet algorithm for about 0.2 ps so that the system is in equilibrium. See Figure 2.

Then the equilibration routine runs in loop for quite some time, around 5 ps to get a system with desired temperature, T_{eq} , and pressure P_{eq} . The routine is as follows.

The instantaneous temperature, $\mathcal{T}(t)$, and pressure, $\mathcal{P}(t)$, is measured accordingly to (2) respectively (3).

$$\mathcal{T}(t) = \frac{2}{3Nk_b} \sum_{i=1}^N \frac{\mathbf{p}_i^2}{2m} \quad (2)$$

$$\mathcal{P}(t) = \frac{1}{3V} \sum_{i=1}^N \left(\frac{\mathbf{p}_i^2}{2m} + \mathbf{r}_i \cdot \mathbf{F}_i \right) \quad (3)$$

Where N is the number of particles, k_b is Boltzmann's constant, V is the volume of the whole system of 256 atoms. The time dependence comes from the dynamical variables \mathbf{p}_i , \mathbf{r}_i and \mathbf{F}_i which is momentum, position and force on each atom.

Next in the algorithm is the rescaling of velocity and position, as seen in (2) a rescaling in velocity will change the temperature. The pressure depends on both velocity and position as seen in (3), but by rescaling only position we will not effect the temperature. The rescaling for position and velocity is done according to (4) respectively (5):

$$\mathbf{v}_i^{\text{new}} = \alpha_T^{1/2} \mathbf{v}_i^{\text{old}}, \quad \alpha_T = 1 + 2 \frac{\Delta t}{\tau_T} \frac{T_{eq} - \mathcal{T}}{\mathcal{T}} \quad (4)$$

$$\mathbf{r}_i^{\text{new}} = \alpha_P^{1/3} \mathbf{r}_i^{\text{old}}, \quad \alpha_P = 1 - \kappa \frac{\Delta t}{\tau_P} (P_{eq} - \mathcal{P}) \quad (5)$$

Where we have Δt as the time step and τ_T & τ_P approximately describes the time for which the system exponentially decays to the desired temperature and pressure, these times have been put to 100 times the time step. κ is the isothermal compressibility

which depends on the volume change with pressure, and we thus let it be constant (the value at 1 atm and 300 K) due to the small volume changes. We have used the value $2.219 \text{ \AA}^3/\text{eV}$ for the isothermal compressibility [2].

It is important to remember that rescaling the positions means that the lattice constant also rescales. All rescalings are done in each time step without relaxing the system, which is motivated by the small changes so the system should be almost in an equilibrated state at all times. And after some time, sufficiently long to have the system with the specified temperature and pressure we go in to production period where we simply turn off the rescaling of position and velocity.

When implementing this routine we can see in Figure 6 for the solid phase that we get quite some oscillation in temperature, around $\pm 75 \text{ K}$. But in the production, where we have turned off the scaling used in the equilibration period, from 50 ps to 60 ps where we average the temperature we get a value of 774.6 K, which is within reasonable distance from our desired temperature 773 K.

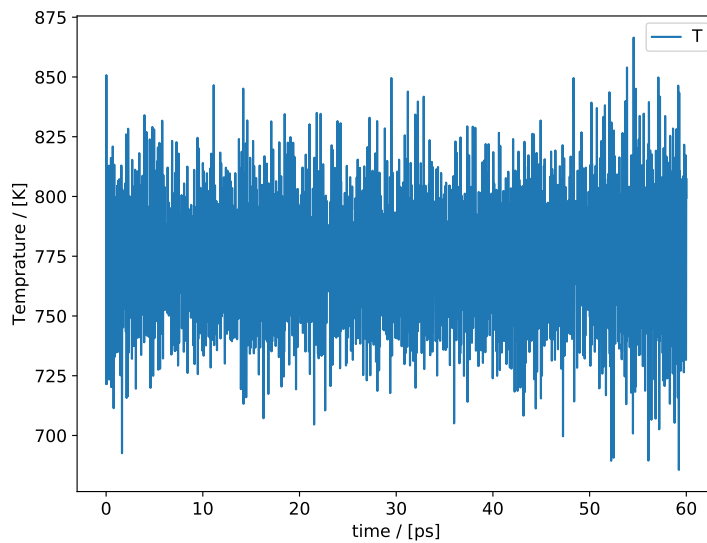


Figure 5: Temperature time evolution for task 3.

The pressure has bigger oscillations around $\pm 0.001 \text{ eV/\AA}^3$, which is around $\pm 0.16 \text{ GPa}$. In the production from 50 ps to 60 ps where we average the pressure we get a value of $0.000056 \text{ eV/\AA}^3$, which is around 9.0 MPa. This is a distance from our desired pressure of 1 bar=0.1 MPa, and shows that it is very hard to equilibrate at small pressures.

One issue which is seen in the figure for the pressure Figure 6 is the big oscillations which makes it hard to say that we have reached equilibrium. If we investigate the lattice constant which as said is rescaled during the equilibration we can see how it stabilizes it self which indicates that we have reached equilibrium. In Figure 7 we can see this stabilization and before going in to production run we make an average of the last 10 ps of the lattice constant which for us was 4.0956 \AA .

In order to convince ourselves that we have solid state we calculated the distance from a position at a specific time. To be more precise for three atoms, when initiating with `init.fcc` atom number 1, 100, and 200, we calculated the distance at each time in the production run to the position they had in the beginning of the production time. This distances as functions of time is seen in Figure 8 where we can see an oscillating behavior which indicates a solid phase.

Task 4

To equilibrate to liquid phase with temperature 700°C and pressure 1 Bar the procedure is very similar to that in Task 4 The big difference is that in order to get the system melted

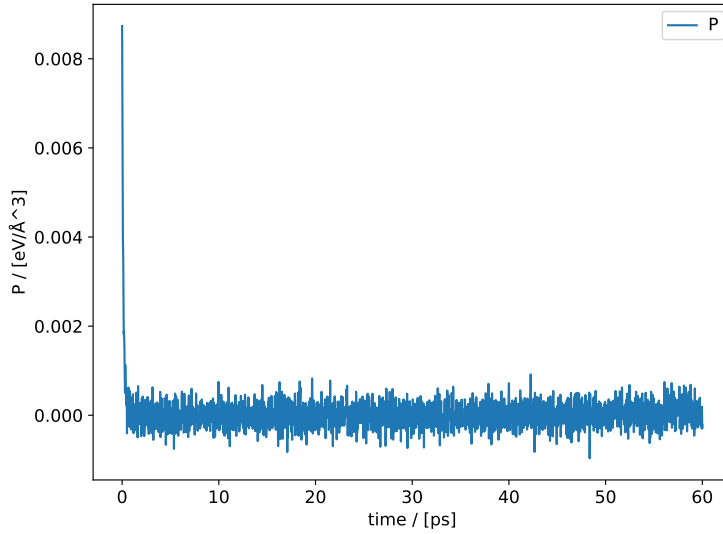


Figure 6: Pressure time evolution for task 3.

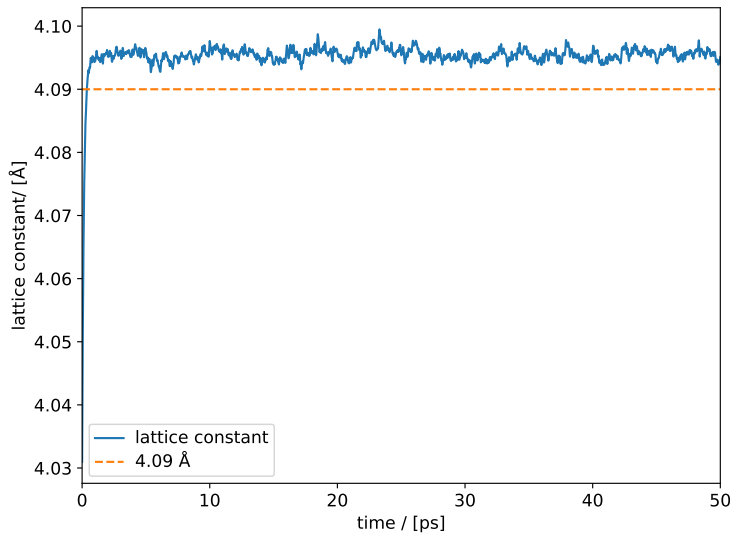


Figure 7: Lattice constant time evolution for task 3.

it is preferable to have melting period which is exactly the same as the equilibration period but with a higher temperature we used 1500 °C.

We can see here as in task 3 that we have some oscillation for the temperature, ± 75 K. But for the production time from 60 ps to 70 ps when we average the temperature we get 970.7 K, which is reasonable closed to the target temperature 973 K.

The pressure got bigger oscillations than the temperature, of around ± 0.0015 eV/Å³ which is in "ordinary" units ± 0.24 GPa. But in the production run the last 10 ps we average the pressure and get 0.000 035 eV/Å³, in "ordinary" units 5.6 MPa which in .

As we said we have quite big oscillations in the pressure time evolution and thus it can be hard to claim that we have reached equilibrium. But if we instead investigate the lattice constant we can see in Figure 11 that it stabilize around 4.264 Å, which is the average for the last 10 ps. And as we have said the lattice constant is rescaled during the equilibration period which is the way to change pressure and thus a stabilized lattice constant is an indicator for reaching equilibrium.

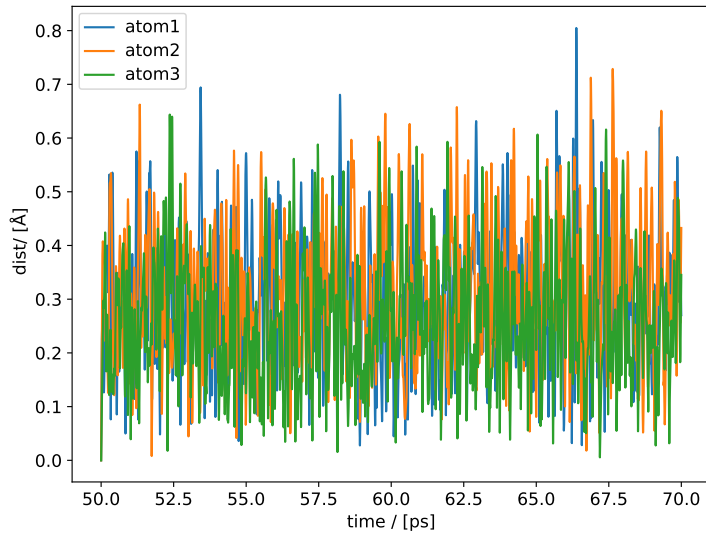


Figure 8: Distance for three atoms, atom 1, 100, & 200, to there respectively first position in the production run. We can see that they oscillate around that position, i.e. no drift so we have solid phase.

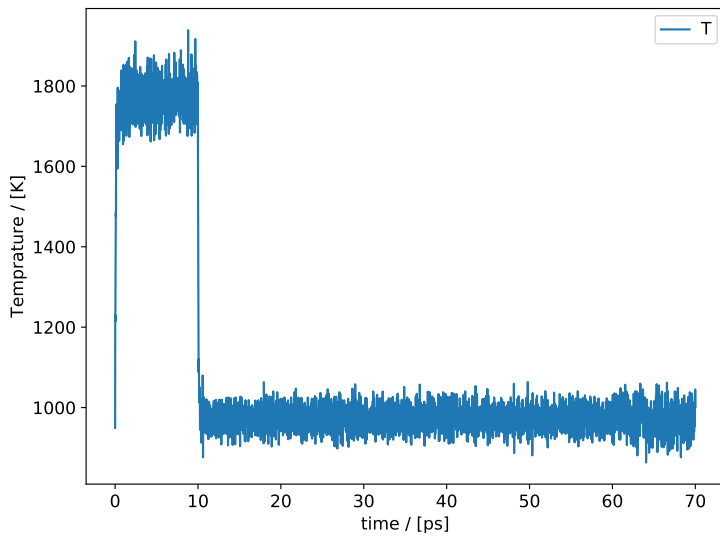


Figure 9: Temperature time evolution for task 4.

In order to convince ourselves that we have a liquid we calculated the distance for three atoms to their respectively position when the production run start at each time step during the production. And in contrast to task 3 with the corresponding Figure 8 where we had oscillations we now have drift instead which indicates a liquid, the drift during 20 ps is up to 10 Å which is more than double of the lattice constant and as such we can conclude that it is not a solid state.

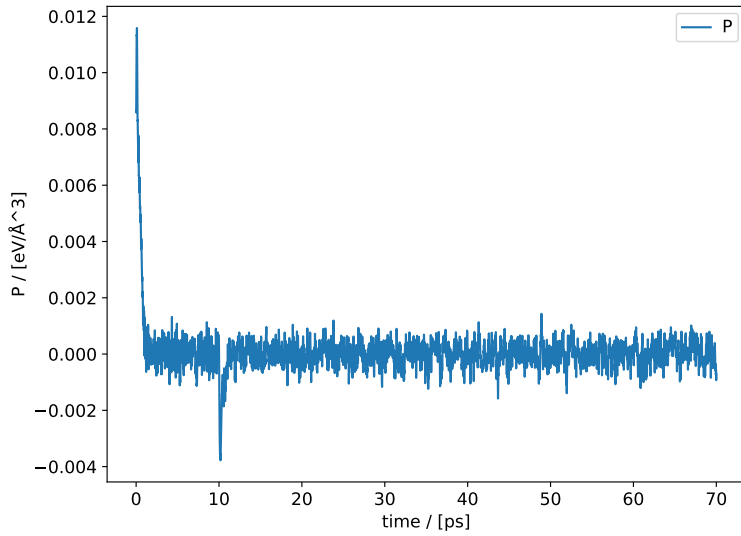


Figure 10: Pressure time evolution for task 4.

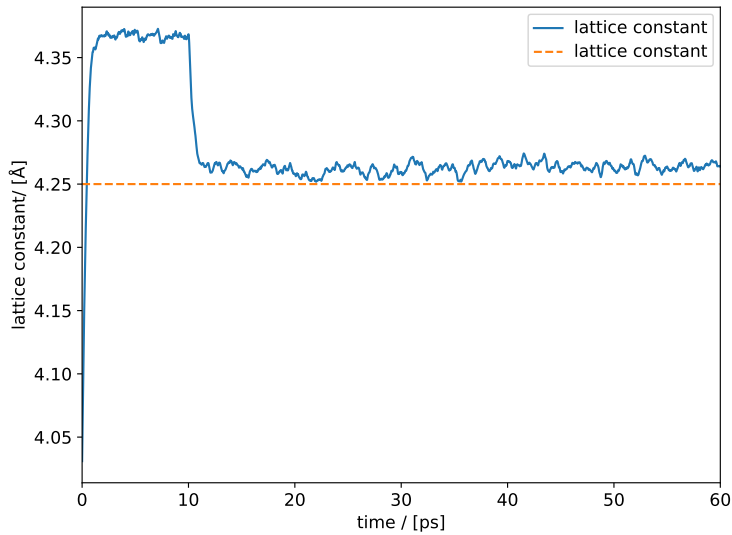


Figure 11: Temperature time evolutions for task 4.

Task 5

In the lecture notes [1], the *mean squared displacement* $\Delta_{\text{MSD}}(t)$ is defined as

$$\Delta_{\text{MSD}}(t) = \left\langle \left\langle (\mathbf{r}_i(t + t') - \mathbf{r}_i(t'))^2 \right\rangle_i \right\rangle_{t'}, \quad (6)$$

where the index i runs over all the particles, and t' runs over some interval $[0, t_{\text{max}}]$.

In Figure 13 and Figure 14, the mean squared displacement is presented for a time interval $[0, 20.0 \text{ ps}]$ for the solid respectively liquid phase.

After an identical equilibration as in task 3 we do a production run, where no rescaling is done that is we do not change the energy in the system, to obtain a serie of positions at difrent times. From this serie we calculate the mean square displacement as in (6).

When doing the same calculation, that of mean square displacement, for the liquid phase the result differs from the solid case. (Now as well the equilibration is the same as

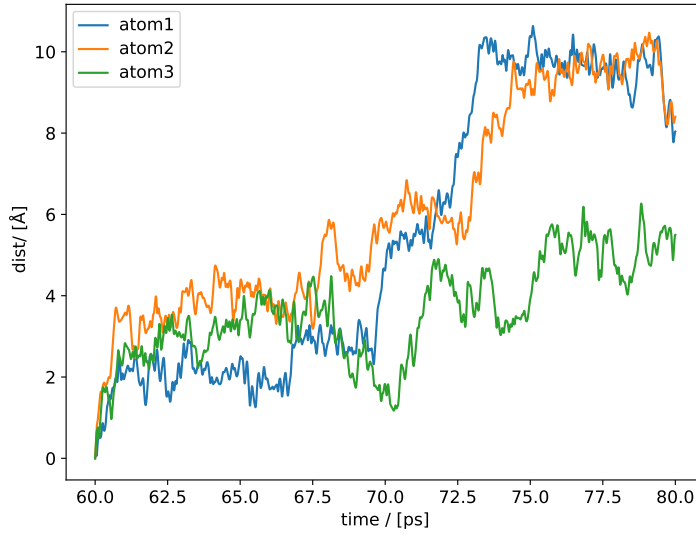


Figure 12: Temperature time evolution for task 4. The reason for keeping the sample at a much higher temperature from the first 10 ps is that we want to be sure that the sample is in a liquid state.

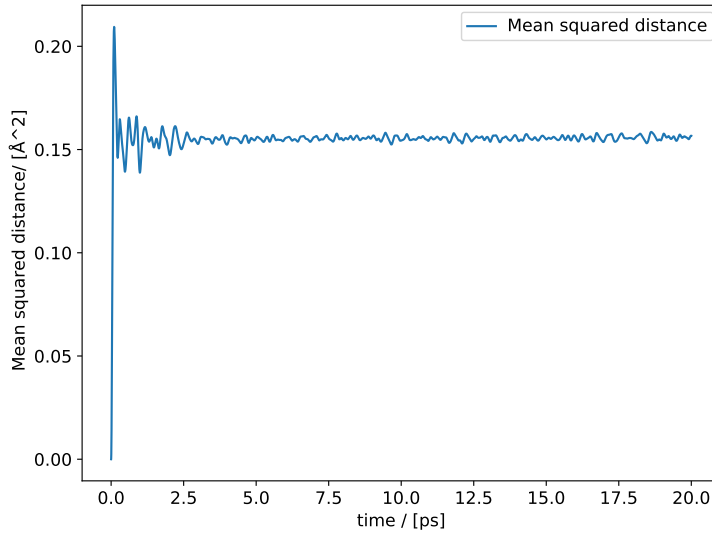


Figure 13: Mean squared displacements of the particles for the solid phase.

in task 4.) Now the mean square displacement continue to grow linearly, which relates to the self diffusion.

Fick's law,

$$\mathbf{j} = -D\nabla n, \quad (7)$$

relates the flux \mathbf{j} to the gradient of the concentration n through the *self-diffusion* coefficient D . By substituting in the continuity equation

$$\frac{\partial}{\partial t}n - \nabla \cdot \mathbf{j} = 0, \quad (8)$$

into (6), we get

$$\frac{\partial}{\partial t}n(\mathbf{r}, t) = D \nabla^2 n(\mathbf{r}, t). \quad (9)$$

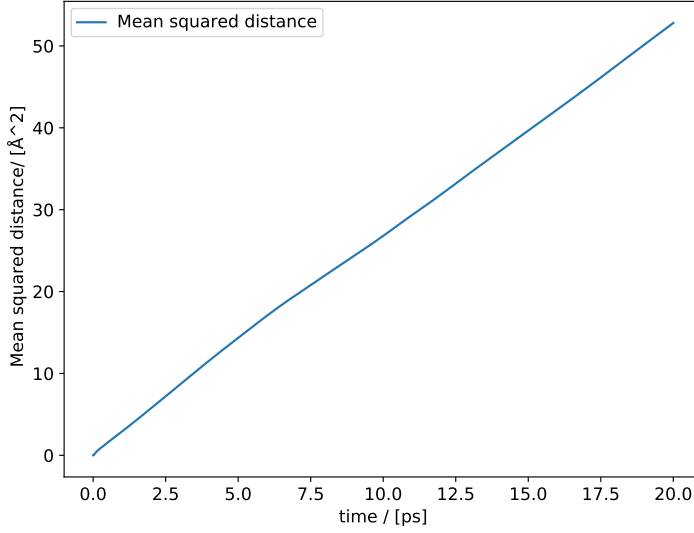


Figure 14: Mean squared displacements of the particles for the liquid phase. Taking the last point on the plot (20, 52.8048) giving the self diffusion coefficient to be $0.44 \text{ Å}^2/\text{ps}$

This is a heat equation with kernel

$$G(\mathbf{r}, t) = \frac{1}{(4\pi Dt)^{\frac{3}{2}}} \exp\left[\frac{-\mathbf{r}^2}{4Dt}\right]. \quad (10)$$

Since (6) is a space average,

$$\Delta_{\text{MSD}}(t) = \int_{\mathbb{R}^3} \int_0^t \mathbf{r}^2 n(\mathbf{r}, t') d^3\mathbf{r} dt' = \int_{\mathbb{R}^3} \int_0^t \mathbf{r}^2 G(\mathbf{r}, t') d^3\mathbf{r} dt' \quad (11)$$

$$= \int_0^\infty \int_0^t r^2 \frac{1}{(4\pi Dt')^{\frac{3}{2}}} \exp\left[\frac{-r^2}{4Dt'}\right] r^2 4\pi dr dt' \xrightarrow{t \rightarrow \infty} 6Dt. \quad (12)$$

Thus, in the limit $t \rightarrow \infty$, the relation

$$\lim_{t \rightarrow \infty} \frac{\Delta_{\text{MSD}}(t)}{6t} = D \quad (13)$$

is obtained.

From the data generating Figure 14, $D \approx 0.44 \text{ Å ps}^{-1}$ was obtained.

Task 6

The *velocity correlation* is defined as

$$\Phi(t) = \left\langle \left\langle \mathbf{v}_i(t + t') \cdot \mathbf{v}_i(t') \right\rangle_{i,t'} \right\rangle. \quad (14)$$

Roughly, it tells you how much knowing a particles velocity at time t will help you know it's velocity at time $t + t'$. From (91) in the lecture notes [1], we have another way to calculate D , namely by

$$D = \frac{1}{6} \lim_{\omega \rightarrow 0} \hat{\Phi}(\omega) \quad (15)$$

where

$$\hat{\Phi}(\omega) = 2 \int_0^\infty \Phi(t) \cos \omega t dt. \quad (16)$$

The naive way of doing this is let $\omega \rightarrow 0$ inside the integral side, in which case we are left with

$$D = \frac{1}{3} \int_0^\infty \Phi(t) dt. \quad (17)$$

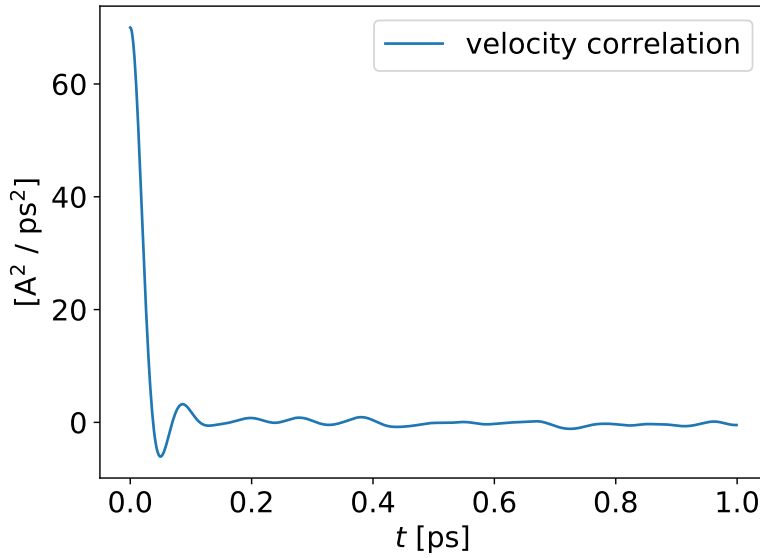


Figure 15: The velocity correlation plotted as a function of time for a liquid aluminum sample at 700 °C.

From this we obtain $D \approx 0.4 \text{ \AA}$. Which agrees with task 5.

In Figure 15, the velocity correlation function is plotted.

Task 7

There is another, possibly faster, way to calculate (14). By using fast Fourier transform (FFT), we can calculate the correlation function S_l of any dynamic quantity $\mathcal{A}(t)$ as

$$S_l = \sum_{k=0}^{N-1} h_{l+k} h_k^* \quad (18)$$

where

$$h_k = \begin{cases} \mathcal{A}(k\Delta t) & , \text{ if } k < N \\ 0 & , \text{ if } k > N \end{cases} \quad (19)$$

and

$$h_k = h_{k+2N}. \quad (20)$$

From the circular correlation theorem,

$$S_l = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \exp 2\pi i l n / N. \quad (21)$$

In Figure 16, the result of this calculation is presented. It is clear that it agrees well with Figure 15.

Concluding discussion

References

- [1] Göran Wahnström, *MOLECULAR DYNAMICS Lecture notes*. Chalmers university of technology, Göteborg, 30 Oktober 2019, available at <https://chalmers.instructure.com/courses/7636/modules> (2019-11-22).
- [2] Wolfram|Alpha, Wolfram Alpha LLC. <https://www.wolframalpha.com/input/?i=%28aluminum+bulk+modulus%29%5E-1+in+%C3%A5ngstr%C3%B6m%5E3+%2F+eV> (2019-12-03).

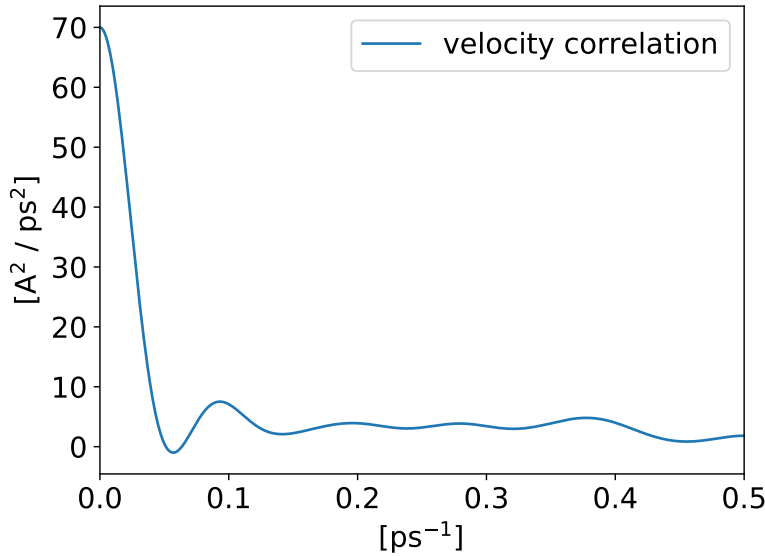


Figure 16: Plot of velocity correlation calculated from (21).

A Source Code

Include all source code here in the appendix. Keep the code formatting clean, use indentation, and comment your code to make it easy to understand. Also, break lines that are too long. (Keep them under 80 characters!)

A.1 Main file MD_main.c

```

1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  #include <time.h>
5  #include <string.h>
6  #include "initfcc.h"
7  #include "alpotential.h"
8  #include "verlet_funcs.h"
9  #include "utility_funcs.h"
10 #include "fft_func.h"
11
12 /* define some shorthands */
13 #define kB (8.617e-5) // Boltzmann constant [eV / K]
14 #define AL_MASS 0.0027964394 // [eV ps^2 / A^2]
15 #define Nc 4
16 #define NBR_CELLS (Nc * Nc * Nc)
17 #define NBR_ATOMS (4 * NBR_CELLS)
18 #define a0_LIST_LENGTH 45
19
20 /* declare some global variables */
21 double a0;
22
23
24 void task1() {
25     double positions[NBR_ATOMS][3]; // [A]
26     double a0; // [A]
27     double a0_list[a0_LIST_LENGTH];
28     double potential_energy; // [eV]
29
30     /*declare variable to sum over */
31     int i;
32
33     FILE *potential_energy_file;
34     FILE *a0_file;
35
36     /* Assign list of unit cell lengths */
37     for (i = 0; i < a0_LIST_LENGTH; i++) {
38         a0_list[i] = pow(64.0 + 0.1 * i, 1 / 3.0); // in order to match Fig 1
39     }
40
41     potential_energy_file = fopen("potential_energy.txt", "w");
42     a0_file = fopen("a0.txt", "w");

```

```

43
44 /* Calculate potential energies */
45 for (i = 0; i < a0_LIST_LENGTH; i++) {
46     a0 = a0_list[i];
47     init_fcc(positions, Nc, a0);
48     potential_energy = get_energy_AL(positions, Nc * a0, NBR_ATOMS);
49
50     /* write energies to file */
51     fprintf(potential_energy_file, "%e\n", potential_energy / NBR_CELLS); //←
52     fprintf(a0_file, "%e\n", a0);
53 }
54
55 fclose(potential_energy_file);
56 fclose(a0_file);
57 }
58
59
60 void task2() {
61     double positions [NBR_ATOMS][3]; // [A]
62     double velocities [NBR_ATOMS][3]; // [A / ps]
63     double accelerations [NBR_ATOMS][3]; // [A / ps^2]
64     double forces [NBR_ATOMS][3]; // [eV / A]
65     double disturbance; // [A]
66     int i, j; // variables to sum over
67
68     /* Initiate some variables */
69     int nbr_timesteps = 1e3;
70     double dt = 1e-3; // [ps]
71     a0 = pow(66.0, 1 / 3.0); // [A]
72
73     /* Allocate some memory & declare some files for storing result from Verlet ←
74     algorithm */
75     double *potential_energies = malloc(nbr_timesteps * sizeof(double));
76     double *kinetic_energies = malloc(nbr_timesteps * sizeof(double));
77     FILE *potential_energies_file;
78     FILE *kinetic_energies_file;
79     FILE *parameters_file;
80
81     /* Initiate positions & introduce disturbances */
82     init_fcc(positions, Nc, a0);
83     for (i = 0; i < NBR_ATOMS; i++){
84         for (j = 0; j < 3; j++){
85             disturbance = 2 * 0.065 * (0.5 - (double) rand() / (double) RAND_MAX←
86             ) * a0; // +/-6.5% of a0 in each direction
87             positions[i][j] += disturbance;
88         }
89     }
90
91     /* Initiate velocities & accelerations */
92     get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
93     init_vel_acc(velocities, accelerations, forces);
94
95     /* Perform velocity Verlet algorithm */
96     for (i = 0; i < nbr_timesteps; i++) {
97         /* Calculate & store energies */
98         potential_energies[i] = get_energy_AL(positions, Nc * a0, NBR_ATOMS);
99         kinetic_energies[i] = get_kinetic_energy(velocities, NBR_ATOMS);
100
101         /* Update positions, velocities, & accelerations using forces */
102         verlet_single(positions, velocities, accelerations, forces, dt, a0);
103     }
104
105     /* write energies to file */
106     potential_energies_file = fopen("potential_energies_task2.bin", "w");
107     kinetic_energies_file = fopen("kinetic_energies_task2.bin", "w");
108     parameters_file = fopen("parameters_task2.bin", "w");
109
110     fwrite(potential_energies, sizeof(double), nbr_timesteps, ←
111     potential_energies_file);
112     fwrite(kinetic_energies, sizeof(double), nbr_timesteps, ←
113     kinetic_energies_file);
114     double paramList[4] = {nbr_timesteps, dt, NBR_ATOMS, a0};
115     int nbr_parameters = 4;
116     fwrite(paramList, nbr_parameters * sizeof(double), 1, parameters_file);
117
118     fclose(potential_energies_file);
119     fclose(kinetic_energies_file);
120     fclose(parameters_file);
121
122     free(potential_energies);
123     free(kinetic_energies);
124 }
125
126 void task3() {
127     double positions[NBR_ATOMS][3]; // [A]
128     double velocities [NBR_ATOMS][3]; // [A / ps]
129     double accelerations [NBR_ATOMS][3]; // [A / ps^2]

```

```

129 double forces[NBR_ATOMS][3]; // [eV / A]
130 double disturbance; // [A]
131 double temperature = 0.0; // [K]
132 double pressure = 0.0; // []
133 int i, j; // variables to sum over
134
135 /* Initiate some variables */
136 int nbr_timesteps_per_epoch = 1;
137 int nbr_epochs = 1e4;
138 double dt = 1e-3;
139 a0 = pow(66.0, 1 / 3.0);
140 double T_eq = 273.15 + 500.0; // [K]
141 double P_eq = 6.242e-7; // 1 bar in [eV / A^3]
142 double T_decay_constant = 100.0 * dt; // [ps]
143 double P_decay_constant = 100.0 * dt; // [ps]
144
145 /* Variables for writing to file */
146 double temperatures[nbr_epochs];
147 double pressures[nbr_epochs];
148 double parameters[6] = {nbr_timesteps_per_epoch, nbr_epochs, ←
    T_decay_constant, P_decay_constant, a0, dt};
149 int nbr_parameters = 6;
150 FILE *temperatures_file;
151 FILE *pressures_file;
152 FILE *parameters_file;
153 FILE *final_positions_file;
154 FILE *final_velocities_file;
155
156
157 /* Initiate positions & introduce disturbances */
158 init_fcc(positions, Nc, a0);
159 for (i = 0; i < NBR_ATOMS; i++) {
160     for (j = 0; j < 3; j++) {
161         disturbance = 2 * 0.065 * (0.5 - (double) rand() / (double) RAND_MAX←
            ) * a0; // +/-6.5% of a0 in each direction
162         positions[i][j] += disturbance;
163     }
164 }
165
166 /* Initiate velocities & accelerations */
167 get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
168 init_vel_acc(velocities, accelerations, forces);
169
170 /* Equilibrate the lattice */
171 for (i = 1; i < nbr_epochs + 1; i++) {
172
173     /* Perform one epoch of velocity Verlet algorithm.
174     temperature & pressure are calculated from time averages. */
175     verlet_task3(temperature, pressure, positions, velocities, accelerations←
        , a0, dt, nbr_timesteps_per_epoch);
176
177     /* Update a0, positions, & velocities */
178     temperature = get_temperature(velocities, NBR_ATOMS);
179     pressure = get_pressure(a0, positions, velocities, NBR_ATOMS);
180     equilibration_update(&a0, positions, velocities, temperature, pressure, ←
        T_eq, P_eq, T_decay_constant, P_decay_constant, dt);
181
182     /* Save temperatures & pressures for later */
183     temperatures[i-1] = temperature;
184     pressures[i-1] = pressure;
185 }
186
187 /* Write relevant info to file */
188 temperatures_file = fopen("temperatures_task3.bin", "w"); // to be used in ←
    plot
189 pressures_file = fopen("pressures_task3.bin", "w"); // to be used in plot
190 parameters_file = fopen("parameters_task3.bin", "w"); // to be used in plot
191 final_positions_file = fopen("final_positions_task3.bin", "w"); // to be ←
    used in task 5
192 final_velocities_file = fopen("final_velocities_task3.bin", "w"); // to be ←
    used in task 5
193
194 fwrite(temperatures, sizeof(double), nbr_epochs, temperatures_file);
195 fwrite(pressures, sizeof(double), nbr_epochs, pressures_file);
196 fwrite(parameters, nbr_parameters * sizeof(double), 1, parameters_file);
197 fwrite(positions, NBR_ATOMS * sizeof(double) * 3, 1, final_positions_file);
198 fwrite(velocities, NBR_ATOMS * sizeof(double) * 3, 1, final_velocities_file)←
    ;
199
200 fclose(temperatures_file);
201 fclose(pressures_file);
202 fclose(parameters_file);
203 fclose(final_positions_file);
204 fclose(final_velocities_file);
205 }
206
207
208 void task4() {
209     double positions[NBR_ATOMS][3]; // [A]
210     double velocities[NBR_ATOMS][3]; // [A / ps]
211     double accelerations[NBR_ATOMS][3]; // [A / ps^2]

```

```

212 double forces[NBR_ATOMS][3]; // [eV / A]
213 double disturbance; // [A]
214 double temperature = 0.0; // [K]
215 double pressure = 0.0; // []
216 int i, j; // variables to sum over
217
218 /* Initiate some variables */
219 int nbr_timesteps_per_epoch = 1;
220 int nbr_epochs = 1e4;
221 double dt = 1e-3;
222 a0 = pow(66.0, 1 / 3.0);
223 double T_eq = 273.15 + 700.0; // [K]
224 double T_pre_eq = 2000; // [K] pre-equilibration temperature for melting the
    aluminum
225 double P_eq = 6.242e-7; // [eV / A^3]
226 double T_decay_constant = 100.0 * dt; // [ps]
227 double P_decay_constant = 100.0 * dt; // [ps]
228
229 /* Variables for writing to file */
230 double temperatures[nbr_epochs];
231 double pressures[nbr_epochs];
232 double parameters[6] = {nbr_timesteps_per_epoch, nbr_epochs, T_
    decay_constant, P_decay_constant, a0, dt};
233 FILE *temperatures_file;
234 FILE *pressures_file;
235 FILE *parameters_file;
236 FILE *final_positions_file;
237 FILE *final_velocities_file;
238
239
240 /* Initiate positions & introduce disturbances */
241 init_fcc(positions, Nc, a0);
242 for (i = 0; i < NBR_ATOMS; i++) {
243     for (j = 0; j < 3; j++) {
244         disturbance = 2 * 0.065 * (0.5 - (double) rand() / (double) RAND_MAX)
            * a0; // +/-6.5% of a0 in each direction
245         positions[i][j] += disturbance;
246     }
247 }
248
249 /* Initiate velocities & accelerations */
250 get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
251 init_vel_acc(velocities, accelerations, forces);
252
253 /* Pre-equilibrate the lattice temperature so that we're certian we're in
    liquid phase */
254 for (i = 0; i < nbr_epochs / 10; i++) {
255
256     /* Perform one epoch of velocity Verlet algorithm.
257     temperature & pressure are calculated from time averages. */
258     verlet_task3(temperature, pressure, positions, velocities, accelerations,
        a0, dt, nbr_timesteps_per_epoch);
259
260     /* Update a0, positions, & velocities */
261     temperature = get_temperature(velocities, NBR_ATOMS);
262     pressure = get_pressure(a0, positions, velocities, NBR_ATOMS);
263     equilibration_update(&a0, positions, velocities, temperature, pressure,
        T_pre_eq, P_eq, T_decay_constant, P_decay_constant, dt);
264 }
265
266
267 /* Equilibrate the lattice */
268 for (i = 0; i < nbr_epochs; i++) {
269
270     /* Perform one epoch of velocity Verlet algorithm.
271     temperature & pressure are calculated from time averages. */
272     verlet_task3(temperature, pressure, positions, velocities, accelerations,
        a0, dt, nbr_timesteps_per_epoch);
273
274     /* Update a0, positions, & velocities */
275     temperature = get_temperature(velocities, NBR_ATOMS);
276     pressure = get_pressure(a0, positions, velocities, NBR_ATOMS);
277     equilibration_update(&a0, positions, velocities, temperature, pressure,
        T_eq, P_eq, T_decay_constant, P_decay_constant, dt);
278
279     /* Save temperatures & pressures for later */
280     temperatures[i] = temperature;
281     pressures[i] = pressure;
282 }
283
284 /* Write relevant info to file */
285 temperatures_file = fopen("temperatures_task4.bin", "w"); // to be used in
    plot
286 pressures_file = fopen("pressures_task4.bin", "w"); // to be used in plot
287 parameters_file = fopen("parameters_task4.bin", "w"); // to be used in plot
288 final_positions_file = fopen("final_positions_task4.bin", "w"); // to be
    used in task 5
289 final_velocities_file = fopen("final_velocities_task4.bin", "w"); // to be
    used in task 5
290
291 fwrite(temperatures, sizeof(double), nbr_epochs, temperatures_file);

```

```

292 fwrite(pressures, sizeof(double), nbr_epochs, pressures_file);
293 fwrite(parameters, sizeof(parameters), 1, parameters_file);
294 fwrite(positions, NBR_ATOMS * sizeof(double) * 3, 1, final_positions_file);
295 fwrite(velocities, NBR_ATOMS * sizeof(double) * 3, 1, final_velocities_file);
296
297 fclose(temperatures_file);
298 fclose(pressures_file);
299 fclose(parameters_file);
300 fclose(final_positions_file);
301 fclose(final_velocities_file);
302 }
303
304
305 void task5() {
306     int nbr_timesteps = 1000;
307     double dt = 1e-3; // [ps]
308     double parameters[2] = {dt, nbr_timesteps};
309     int nbr_parameters = 2;
310     int i, j;
311
312     double positions[NBR_ATOMS][3]; // [A]
313     double velocities[NBR_ATOMS][3]; // [A / ps]
314     double accelerations[NBR_ATOMS][3]; // [A / ps^2]
315     double forces[NBR_ATOMS][3]; // [eV / A]
316     double initial_parameters[6];
317
318     /* Allocate some memory for storing in each timestep */
319     double *mean_squared_displacements = malloc(nbr_timesteps * sizeof(double));
320     double ***positions_list = (double ***)malloc(NBR_ATOMS * sizeof(double **));
321     // positions_list[nbr_atoms][3][2 * nbr_timesteps]
322     for (i = 0; i < NBR_ATOMS; i++) {
323         positions_list[i] = (double **)malloc(3 * sizeof(double *));
324         positions_list[i][0] = (double *)malloc(2 * nbr_timesteps * sizeof(
325             double));
326         positions_list[i][1] = (double *)malloc(2 * nbr_timesteps * sizeof(
327             double));
328         positions_list[i][2] = (double *)malloc(2 * nbr_timesteps * sizeof(
329             double));
330     }
331
332     FILE *initial_positions_file;
333     FILE *initial_velocities_file;
334     FILE *initial_parameters_file;
335     FILE *parameters_file;
336     FILE *mean_squared_displacements_file;
337
338     /* Initiate positions & velocities */
339     /* The final positions & velocities from Task 3/4 are tuned to have ←
340        temperature 500/700 C & pressure 1 bar */
341     initial_positions_file = fopen("final_positions_task4.bin", "r");
342     initial_velocities_file = fopen("final_velocities_task4.bin", "r");
343     initial_parameters_file = fopen("parameters_task4.bin", "r");
344
345     int tmp; // this is a bodge way of suppressing warning about fread:s return ←
346     value unused
347     tmp = fread(positions, NBR_ATOMS * sizeof(double) * 3, 1, ←
348         initial_positions_file);
349     tmp = fread(velocities, NBR_ATOMS * sizeof(double) * 3, 1, ←
350         initial_velocities_file);
351     tmp = fread(initial_parameters, nbr_parameters * sizeof(double), 1, ←
352         initial_parameters_file);
353     tmp++;
354
355     fclose(initial_positions_file);
356     fclose(initial_velocities_file);
357     fclose(initial_parameters_file);
358
359     /* Initiate forces & accelerations */
360     a0 = initial_parameters[4];
361     get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
362     for (i = 0; i < NBR_ATOMS; i++) {
363         accelerations[i][0] = forces[i][0] / AL_MASS;
364         accelerations[i][1] = forces[i][1] / AL_MASS;
365         accelerations[i][2] = forces[i][2] / AL_MASS;
366     }
367
368     /* Perform velocity Verlet algorithm */
369     for (i = 1; i < 2 * nbr_timesteps; i++) {
370         /* Update positions, velocities, & accelerations using forces */
371         verlet_single(positions, velocities, accelerations, forces, dt, a0);
372
373         /* Store positions */
374         for (j = 0; j < NBR_ATOMS; j++) {
375             positions_list[j][0][i] = positions[j][0];
376             positions_list[j][1][i] = positions[j][1];
377             positions_list[j][2][i] = positions[j][2];
378         }
379     }
380 }

```



```

373  /* Calculate mean squared displacements */
374  for (i = 0; i < nbr_timesteps; i++) {
375      mean_squared_displacements[i] = mean_squared_displacement(i, ←
          positions_list, dt, nbr_timesteps);
376  }
377
378  /* Write mean squared displacement to file */
379  mean_squared_displacements_file = fopen("mean_squared_displacements_task5.←
      bin", "w");
380  parameters_file = fopen("parameters_task5.bin", "w");
381
382  fwrite(mean_squared_displacements, nbr_timesteps * sizeof(double), 1, ←
      mean_squared_displacements_file);
383  fwrite(parameters, nbr_parameters * sizeof(double), 1, parameters_file);
384
385  fclose(mean_squared_displacements_file);
386  fclose(parameters_file);
387
388  free(mean_squared_displacements);
389  free(positions_list);
390  }
391
392
393  void task6() {
394      int nbr_timesteps = 1000;
395      double dt = 1e-3; // [ps]
396      double parameters[2] = {dt, nbr_timesteps};
397      int i, j;
398
399      double positions[NBR_ATOMS][3]; // [A]
400      double velocities[NBR_ATOMS][3]; // [A / ps]
401      double accelerations[NBR_ATOMS][3]; // [A / ps^2]
402      double forces[NBR_ATOMS][3]; // [eV / A]
403      double initial_parameters[6];
404      int nbr_initial_parameters = 6;
405      int nbr_parameters = 6;
406
407      /* Allocate some memory for storing in each timestep */
408      double *velocity_correlations = malloc(nbr_timesteps * sizeof(double));
409      double ***velocities_list = (double ***)malloc(NBR_ATOMS * sizeof(double ***) ←
          ); // velocities_list[nbr_atoms][3][2 * nbr_timesteps]
410      for (i = 0; i < NBR_ATOMS; i++) {
411          velocities_list[i] = (double **)malloc(3 * sizeof(double *));
412          velocities_list[i][0] = (double *)malloc(2 * nbr_timesteps * sizeof(←
              double));
413          velocities_list[i][1] = (double *)malloc(2 * nbr_timesteps * sizeof(←
              double));
414          velocities_list[i][2] = (double *)malloc(2 * nbr_timesteps * sizeof(←
              double));
415      }
416
417      FILE *initial_positions_file;
418      FILE *initial_velocities_file;
419      FILE *initial_parameters_file;
420      FILE *velocities_file;
421      FILE *velocity_correlation_file;
422      FILE *parameters_file;
423
424      /* Initiate positions & velocities */
425      /* The final positions & velocities from Task 3/4 are tuned to have ←
          temperature 500/700 C & pressure 1 bar */
426      initial_positions_file = fopen("final_positions_task4.bin", "r");
427      initial_velocities_file = fopen("final_velocities_task4.bin", "r");
428      initial_parameters_file = fopen("parameters_task4.bin", "r");
429
430      int tmp; // this is a bodgy way of suppressing warning about fread:s return ←
          value unused
431      tmp = fread(positions, NBR_ATOMS * sizeof(double) * 3, 1, ←
          initial_positions_file);
432      tmp = fread(velocities, NBR_ATOMS * sizeof(double) * 3, 1, ←
          initial_velocities_file);
433      tmp = fread(initial_parameters, nbr_initial_parameters * sizeof(double), 1, ←
          initial_parameters_file);
434      tmp++;
435
436      fclose(initial_positions_file);
437      fclose(initial_velocities_file);
438      fclose(initial_parameters_file);
439
440
441      /* Initiate forces & accelerations */
442      a0 = initial_parameters[4];
443      get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
444      for (i = 0; i < NBR_ATOMS; i++) {
445          accelerations[i][0] = forces[i][0] / A1_MASS;
446          accelerations[i][1] = forces[i][1] / A1_MASS;
447          accelerations[i][2] = forces[i][2] / A1_MASS;
448      }
449
450      /* Perform velocity Verlet algorithm */
451      for (i = 1; i < 2 * nbr_timesteps; i++) {

```

```

452     /* Update positions, velocities, & accelerations using forces */
453     verlet_single(positions, velocities, accelerations, forces, dt, a0);
454
455     /* Store velocities */
456     for (j = 0; j < NBR_ATOMS; j++) {
457         velocities_list[j][0][i] = velocities[j][0];
458         velocities_list[j][1][i] = velocities[j][1];
459         velocities_list[j][2][i] = velocities[j][2];
460     }
461 }
462
463 /* Calculate velocity_correlation */
464 for (i = 0; i < nbr_timesteps; i++) {
465     velocity_correlations[i] = velocity_correlation(i, velocities_list, dt, ←
        nbr_timesteps);
466 }
467
468 /* Write velocity correlation to file */
469 velocities_file = fopen("velocities_task6.bin", "w");
470 velocity_correlation_file = fopen("velocity_correlations_task6.bin", "w");
471 parameters_file = fopen("parameters_task6.bin", "w");
472
473 for (i = 0; i < NBR_ATOMS; i++) { // we have to write each list by itself to ←
    file
474     for (j = 0; j < 3; j++) {
475         fwrite(velocities_list[i][j], nbr_timesteps * 2 * sizeof(double), 1, ←
            velocities_file);
476     }
477 }
478 fwrite(velocity_correlations, nbr_timesteps * sizeof(double), 1, ←
    velocity_correlation_file);
479 fwrite(parameters, nbr_parameters * sizeof(double), 1, parameters_file);
480
481 fclose(velocities_file);
482 fclose(velocity_correlation_file);
483 fclose(parameters_file);
484
485 free(velocity_correlations);
486 free(velocities_list);
487 }
488
489 void task7() {
490     int nbr_timesteps = 1000; // ought to be the same as in task 6 (BODGE)
491     double dt = 1e-3; // [ps] ought to be the same as in task 6 (BODGE)
492     // double parameters[2] = {dt, nbr_timesteps};
493     int i, j, k; // for summing over
494
495     /* Allocate some memory for storing in each timestep */
496     // double *velocity_correlations = malloc(nbr_timesteps * sizeof(double)); ←
497     double ***velocities_list = (double ***)malloc(NBR_ATOMS * sizeof(double **)) ←
498     ); // velocities_list[nbr_atoms][3][2 * nbr_timesteps]
499     for (i = 0; i < NBR_ATOMS; i++) {
500         velocities_list[i] = (double **)malloc(3 * sizeof(double *));
501         velocities_list[i][0] = (double *)malloc(2 * nbr_timesteps * sizeof(←
            double));
502         velocities_list[i][1] = (double *)malloc(2 * nbr_timesteps * sizeof(←
            double));
503         velocities_list[i][2] = (double *)malloc(2 * nbr_timesteps * sizeof(←
            double));
504     }
505
506     /* arrays for FFT */
507     double *tmp_array = malloc(nbr_timesteps * sizeof(double));
508     double *powspec = malloc(nbr_timesteps * sizeof(double));
509     double *total_powspec = malloc(nbr_timesteps * sizeof(double));
510     for (k = 0; k < NBR_ATOMS; k++) { // initiate total_powspec
511         total_powspec[k] = 0.0;
512     }
513
514     double *freq = malloc(nbr_timesteps * sizeof(double));
515
516     FILE *velocities_file;
517     FILE *powerspectrum_file;
518     FILE *frequencies_file;
519
520     /* Read velocities from task 6 from file */
521     velocities_file = fopen("velocities_task6.bin", "r");
522     int tmp;
523     for (i = 0; i < NBR_ATOMS; i++) {
524         for (j = 0; j < 3; j++) {
525             tmp = fread(velocities_list[i][j], nbr_timesteps * 2 * sizeof(double) ←
526                 , 1, velocities_file);
527         }
528     }
529     tmp++;
530     fclose(velocities_file);
531
532     /* make FFT (powerspectrum) */
533     for (i = 0; i < NBR_ATOMS; i++) {

```

```

534     for(j = 0; j < 3; j++) {
535         tmp_array = velocities_list[i][j];
536         powerspectrum(tmp_array, powspec, nbr_timesteps); // fft is same as ←
                    inverse fft here since our function is even
537         for(k = 0; k < NBR_ATOMS; k++) {
538             total_powspec[k] += powspec[k] / NBR_ATOMS;
539         }
540     }
541 }
542 // powerspectrum_shift(total_powspec, nbr_timesteps);
543 fft_freq(freq, dt, nbr_timesteps);
544 // fft_freq_shift(freq, dt, nbr_timesteps);
545
546 /* save powerspectrum for plotting */
547 powerspectrum_file = fopen("powerspectrum_task7.bin", "w");
548 frequencies_file = fopen("frequencies_task7.bin", "w");
549
550 fwrite(total_powspec, nbr_timesteps * sizeof(double), 1, powerspectrum_file)←
551 ;
552 fwrite(freq, nbr_timesteps * sizeof(double), 1, frequencies_file);
553 }
554
555 /* Main program */
556 int main()
557 {
558     /* Initiate RNG */
559     srand(time(NULL));
560
561     // task1();
562
563     // task2();
564
565     // task3();
566
567     // task4();
568
569     // task5();
570
571     // task6();
572
573     task7();
574
575     return 0;
576 }
577

```

A.2 Verlet algorithms file verlet_funcs.c

```

1  #include "verlet_funcs.h"
2
3  #include <math.h>
4  #include "utility_funcs.h"
5  #include "alpotential.h"
6
7  /* define some shorthands (same as in main) */
8  #define kB (8.617e-5) // Boltzmann constant [eV / K]
9  #define A1_MASS 0.0027964394 // [eV ps^2 / A^2]
10 #define Nc 4
11 #define NBR_CELLS (Nc * Nc * Nc)
12 #define NBR_ATOMS (4 * NBR_CELLS)
13 #define a0_LIST_LENGTH 45
14
15
16 /* Shorthand for initiating velocities & accelerations */
17 void init_vel_acc(double velocities[][3], double accelerations[][3], double ←
    forces[][3]) {
18     int i;
19     for (i = 0; i < NBR_ATOMS; i++) {
20         velocities[i][0] = 0.0;
21         velocities[i][1] = 0.0;
22         velocities[i][2] = 0.0;
23         accelerations[i][0] = forces[i][0] / A1_MASS;
24         accelerations[i][1] = forces[i][1] / A1_MASS;
25         accelerations[i][2] = forces[i][2] / A1_MASS;
26     }
27 }
28
29
30 /*
31 Run Verlet algorithm 1 time.
32 Updates positions, velocities, & accelerations using volicities, ←
    accelerations, forces, & timestep.
33 */
34 void verlet_single(double positions[][3], double velocities[][3], double ←
    accelerations[][3], double forces[][3], double timestep, double a0) {

```

```

35 int i;
36 for (i = 0; i < NBR_ATOMS; i++) {
37     /* v(t+dt/2) */
38     velocities[i][0] += timestep * 0.5 * accelerations[i][0];
39     velocities[i][1] += timestep * 0.5 * accelerations[i][1];
40     velocities[i][2] += timestep * 0.5 * accelerations[i][2];
41
42     /* q(t+dt) */
43     positions[i][0] += timestep * velocities[i][0];
44     positions[i][1] += timestep * velocities[i][1];
45     positions[i][2] += timestep * velocities[i][2];
46 }
47
48 /* Update forces */
49 get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
50
51 for (i = 0; i < NBR_ATOMS; i++) {
52     /* a(t+dt) */
53     accelerations[i][0] = forces[i][0] / A1_MASS;
54     accelerations[i][1] = forces[i][1] / A1_MASS;
55     accelerations[i][2] = forces[i][2] / A1_MASS;
56
57     /* v(t+dt) */
58     velocities[i][0] += timestep * 0.5 * accelerations[i][0];
59     velocities[i][1] += timestep * 0.5 * accelerations[i][1];
60     velocities[i][2] += timestep * 0.5 * accelerations[i][2];
61 }
62 }
63
64 /*
65  Run Verlet algorithm nbr_timesteps times.
66  Updates positions, velocities, & accelerations using forces & timestep.
67 */
68 void verlet_many(double positions[][3], double velocities[][3], double ←
69     accelerations[][3], double a0, double timestep, int nbr_timesteps) {
70     /* Initiate forces */
71     double forces[NBR_ATOMS][3];
72     get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
73
74     /* Run Verlet algorithm */
75     int i;
76     for (i = 0; i < nbr_timesteps; i++){
77         /* Update positions, velocities, & accelerations using forces */
78         verlet_single(positions, velocities, accelerations, forces, timestep, a0←
79             );
80
81         /* Update forces */
82         get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
83     }
84 }
85
86 /*
87  Run Verlet algorithm nbr_timesteps times.
88  Updates positions, velocities, & accelerations using forces & timestep.
89  Stores time averages of temperature & pressure in temperature_average & ←
90     pressure_average.
91 */
92 void verlet_task3(double temperature_average, double pressure_average, double ←
93     positions[][3], double velocities[][3], double accelerations[][3], double ←
94     a0, double timestep, int nbr_timesteps) {
95     double temperature;
96     temperature_average = 0.0;
97     double pressure;
98     pressure_average = 0.0;
99
100     /* Initiate forces */
101     double forces[NBR_ATOMS][3];
102     get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
103
104     /* Run Verlet algorithm */
105     int i;
106     for (i = 0; i < nbr_timesteps; i++){
107         /* Update positions, velocities, & accelerations using forces */
108         verlet_single(positions, velocities, accelerations, forces, timestep, a0←
109             );
110
111         /* Update temperature & pressure time averages */
112         temperature = get_temperature(velocities, NBR_ATOMS);
113         pressure = get_pressure(a0, positions, velocities, NBR_ATOMS);
114         temperature_average += temperature / (double) nbr_timesteps;
115         pressure_average += pressure / nbr_timesteps;
116     }
117 }

```

A.3 Utility functions file utility_funcs.c

```

1  #include "utility_funcs.h"
2
3  #include <stdio.h>
4  #include <math.h>
5  #include "verlet_funcs.h"
6  #include "alpotential.h"
7
8  /* define some shorthands (same as in main) */
9  #define kB (8.617e-5) // Boltzmann constant [eV / K]
10 #define AL_MASS 0.0027964394 // [eV ps^2 / A^2]
11 #define Nc 4
12 #define NBR_CELLS (Nc * Nc * Nc)
13 #define NBR_ATOMS (4 * NBR_CELLS)
14
15
16 double square(double a) {
17     return a * a;
18 }
19
20
21 /* Returns total kinetic energy of supercell */
22 double get_kinetic_energy(double velocities[][3], int velocities_length) {
23     /* Declare variable to sum over */
24     int i;
25
26     /* Initiate sum of kinetic energies */
27     double sum = 0.0;
28
29     for (i = 0; i < velocities_length; i++){
30         sum += 0.5 * pow(velocities[i][0], 2) * AL_MASS;
31         sum += 0.5 * pow(velocities[i][1], 2) * AL_MASS;
32         sum += 0.5 * pow(velocities[i][2], 2) * AL_MASS;
33     }
34
35     return sum;
36 }
37
38
39 /* Returns temperature of supercell */
40 double get_temperature(double velocities[][3], int velocities_length) {
41     double average_kinetic_energy;
42     double temperature;
43
44     average_kinetic_energy = get_kinetic_energy(velocities, velocities_length) / ←
45     velocities_length;
46     temperature = average_kinetic_energy * 2.0 / (3.0 * kB);
47
48     return temperature;
49 }
50
51 /* Returns pressure of supercell */
52 double get_pressure(double a0, double positions[][3], double velocities[][3], ←
53     int positions_length) {
54     double volume = a0 * a0 * a0 * Nc * Nc * Nc;
55
56     return ((2.0 / 3.0) * get_kinetic_energy(velocities, positions_length) / ←
57         NBR_ATOMS + get_virial_AL(positions, a0 * Nc, NBR_ATOMS)) / volume;
58 }
59
60 /* Updates a0, positions, & velocities using temperature & pressure to better ←
61     match T_eq & P_eq */
62 void equilibration_update(double *a0, double positions[][3], double velocities←
63     [][3], double temperature, double pressure, double T_eq, double P_eq, ←
64     double T_decay_constant, double P_decay_constant, double timestep) {
65     double alpha_T;
66     double alpha_P;
67     double kappa_T = 2.1; // [A^3 / eV] isothermal compressibility of aluminum @←
68     300 K
69     int i; // variable to sum over
70
71     /* Calculate rescaling factors */
72     alpha_T = 1.0 + 2.0 * timestep * (T_eq - temperature) / (T_decay_constant * ←
73     temperature);
74     alpha_P = 1.0 - kappa_T * timestep * (P_eq - pressure) / P_decay_constant;
75
76     /* Rescale positions & velocities in order to get closer to P_eq & T_eq */
77     for (i = 0; i < NBR_ATOMS; i++) {
78         velocities[i][0] *= pow(alpha_T, 1 / 2.0);
79         velocities[i][1] *= pow(alpha_T, 1 / 2.0);
80         velocities[i][2] *= pow(alpha_T, 1 / 2.0);
81
82         positions[i][0] *= pow(alpha_P, 1 / 3.0);
83         positions[i][1] *= pow(alpha_P, 1 / 3.0);
84         positions[i][2] *= pow(alpha_P, 1 / 3.0);
85     }
86     *a0 *= pow(alpha_P, 1 / 3.0);
87 }
88

```

```

83
84 /* Returns mean squared displacement at time t of particles in lattice given an ↵
      array
85 positions_list[nbr_atoms][nbr_dimensions][2 * nbr_timesteps] */
86 double mean_squared_displacement(int t_index, double ***positions_list, double ↵
      timestep, int nbr_timesteps) {
87     int i, j; // variables to sum over
88     double squared_displacements = 0.0; // tmp variable to hold sum
89
90     /* sum up squared displacements over time and particles */
91     for (i = 0; i < nbr_timesteps; i++) {
92         for (j = 0; j < NBR_ATOMS; j++) {
93             squared_displacements += square(positions_list[j][0][i + t_index] - ↵
              positions_list[j][0][i]);
94             squared_displacements += square(positions_list[j][1][i + t_index] - ↵
              positions_list[j][1][i]);
95             squared_displacements += square(positions_list[j][2][i + t_index] - ↵
              positions_list[j][2][i]);
96         }
97     }
98
99     return squared_displacements / (double) (NBR_ATOMS * nbr_timesteps); // ↵
      average by dividing by nbr_atoms & nbr_timesteps
100 }
101
102
103 double velocity_correlation(int t_index, double ***velocities_list, double ↵
      timestep, int nbr_timesteps) {
104     int i, j; // variables to sum over
105     double vel_corr = 0.0; // tmp variable to hold sum
106
107     /* sum up squared displacements over time and particles */
108     for (i = 0; i < nbr_timesteps; i++) {
109         for (j = 0; j < NBR_ATOMS; j++) {
110             vel_corr += velocities_list[j][0][i + t_index] * velocities_list[j]↵
              [0][i];
111             vel_corr += velocities_list[j][1][i + t_index] * velocities_list[j]↵
              [1][i];
112             vel_corr += velocities_list[j][2][i + t_index] * velocities_list[j]↵
              [2][i];
113         }
114     }
115
116     return vel_corr / (double) (NBR_ATOMS * nbr_timesteps); // average by ↵
      dividing by nbr_atoms & nbr_timesteps
117 }

```

A.4 Aluminum properties file alpotential.c

```

1  /*
2  alpotential.c
3  Program that contains functions that calculate properties (potential energy, ↵
      forces, etc.) of a set of Aluminum atoms using an embedded atom model (EAM↵
      ) potential.
4  Created by Anders Lindman on 2013-03-14.
5  */
6
7
8  #include <stdio.h>
9  #include <math.h>
10 #include <stdlib.h>
11
12 /*Parameters for the AL EAM potential */
13 #define PAIR_POTENTIAL_ROWS 18
14 const double pair_potential[90] = {2.0210, 2.2730, 2.4953, 2.7177, 2.9400, ↵
      3.1623, 3.3847, 3.6070, 3.8293, 4.0517, 4.2740, 4.4963, 4.7187, 4.9410, ↵
      5.1633, 5.3857, 5.6080, 6.0630, 2.0051, 0.7093, 0.2127, 0.0202, -0.0386, ↵
      -0.0492, -0.0424, -0.0367, -0.0399, -0.0574, -0.0687, -0.0624, -0.0492, ↵
      -0.0311, -0.0153, -0.0024, -0.0002, 0, -7.2241, -3.3383, -1.3713, -0.4753, ↵
      -0.1171, 0.0069, 0.0374, 0.0122, -0.0524, -0.0818, -0.0090, 0.0499, 0.0735, ↵
      0.0788, 0.0686, 0.0339, -0.0012, 0, 9.3666, 6.0533, 2.7940, 1.2357, ↵
      0.3757, 0.1818, -0.0445, -0.0690, -0.2217, 0.0895, 0.2381, 0.0266, 0.0797, ↵
      -0.0557, 0.0097, -0.1660, 0.0083, 0, -4.3827, -4.8865, -2.3363, -1.2893, ↵
      -0.2907, -0.3393, -0.0367, -0.2290, 0.4667, 0.2227, -0.3170, 0.0796, ↵
      -0.2031, 0.0980, -0.2634, 0.2612, -0.0102, 0};
15
16
17 #define ELECTRON_DENSITY_ROWS 15
18 const double electron_density[75] = {2.0210, 2.2730, 2.5055, 2.7380, 2.9705, ↵
      3.2030, 3.4355, 3.6680, 3.9005, 4.1330, 4.3655, 4.5980, 4.8305, 5.0630, ↵
      6.0630, 0.0824, 0.0918, 0.0883, 0.0775, 0.0647, 0.0512, 0.0392, 0.0291, ↵
      0.0186, 0.0082, 0.0044, 0.0034, 0.0027, 0.0025, 0.0000, 0.0707, 0.0071, ↵
      -0.0344, -0.0533, -0.0578, -0.0560, -0.0465, -0.0428, -0.0486, -0.0318, ↵
      -0.0069, -0.0035, -0.0016, -0.0008, 0, -0.1471, -0.1053, -0.0732, -0.0081, ↵
      -0.0112, 0.0189, 0.0217, -0.0056, -0.0194, 0.0917, 0.0157, -0.0012, 0.0093, ↵

```

```

19     -0.0059, 0, 0.0554, 0.0460, 0.0932, -0.0044, 0.0432, 0.0040, -0.0392, ←
20     -0.0198, 0.1593, -0.1089, -0.0242, 0.0150, -0.0218, 0.0042, 0};
21
22 #define EMBEDDING_ENERGY_ROWS 13
23
24 const double embedding_energy[65] = {0, 0.1000, 0.2000, 0.3000, 0.4000, 0.5000, ←
25     0.6000, 0.7000, 0.8000, 0.9000, 1.0000, 1.1000, 1.2000, 0, -1.1199, ←
26     -1.4075, -1.7100, -1.9871, -2.2318, -2.4038, -2.5538, -2.6224, -2.6570, ←
27     -2.6696, -2.6589, -2.6358, -18.4387, -5.3706, -2.3045, -3.1161, -2.6175, ←
28     -2.0666, -1.6167, -1.1280, -0.4304, -0.2464, -0.0001, 0.1898, 0.2557, ←
29     86.5178, 44.1632, -13.5018, 5.3853, -0.3996, 5.9090, -1.4103, 6.2976, ←
30     0.6785, 1.1611, 1.3022, 0.5971, 0.0612, -141.1819, -192.2166, 62.9570, ←
31     -19.2831, 21.0288, -24.3978, 25.6930, -18.7304, 1.6087, 0.4704, -2.3503, ←
32     -1.7862, -1.7862};
33
34 /* Evaluates the spline in x. */
35 double splineEval(double x, const double *table, int m) {
36     /* int m = mxGetM(spline), i, k; */
37     int i, k;
38
39     /*double *table = mxGetPr(spline);*/
40     double result;
41
42     int k_lo = 0, k_hi = m;
43
44     /* Find the index by bisection. */
45     while (k_hi - k_lo > 1) {
46         k = (k_hi + k_lo) >> 1;
47         if (table[k] > x)
48             k_hi = k;
49         else
50             k_lo = k;
51     }
52
53     /* Switch to local coord. */
54     x -= table[k_lo];
55
56     /* Horner's scheme */
57     result = table[k_lo + 4*m];
58     for (i = 3; i > 0; i--) {
59         result *= x;
60         result += table[k_lo + i*m];
61     }
62
63     return result;
64 }
65
66 /* Evaluates the derivative of the spline in x. */
67 double splineEvalDiff(double x, const double *table, int m) {
68     /*int m = mxGetM(spline), i, k;
69     double *table = mxGetPr(spline);
70     */
71     int i, k;
72     double result;
73
74     int k_lo = 0, k_hi = m;
75
76     /* Find the index by bisection. */
77     while (k_hi - k_lo > 1) {
78         k = (k_hi + k_lo) >> 1;
79         if (table[k] > x)
80             k_hi = k;
81         else
82             k_lo = k;
83     }
84
85     /* Switch to local coord. */
86     x -= table[k_lo];
87
88     /* Horner's scheme */
89     result = 3*table[k_lo + 4*m];
90     for (i = 3; i > 1; i--) {
91         result *= x;
92         result += (i-1)*table[k_lo + i*m];
93     }
94
95     return result;
96 }
97
98 /* Returns the forces */
99 void get_forces_AL(double forces[][3], double positions[][3], double cell_length←
100     , int nbr_atoms)
101 {
102     int i, j;
103     double cell_length_inv, cell_length_sq;
104     double rcut, rcut_sq;
105     double densityi, dens, drho_dr, force;
106     double dUpair_dr;
107     double sxi, syi, szi, sxij, syij, szij, rij, rij_sq;
108
109     double *sx = malloc(nbr_atoms * sizeof (double));

```

```

99 double *sy = malloc(nbr_atoms * sizeof (double));
100 double *sz = malloc(nbr_atoms * sizeof (double));
101 double *fx = malloc(nbr_atoms * sizeof (double));
102 double *fy = malloc(nbr_atoms * sizeof (double));
103 double *fz = malloc(nbr_atoms * sizeof (double));
104
105 double *density = malloc(nbr_atoms * sizeof (double));
106 double *dUembed_drho = malloc(nbr_atoms * sizeof (double));
107
108 rcut = 6.06;
109 rcut_sq = rcut * rcut;
110
111 cell_length_inv = 1 / cell_length;
112 cell_length_sq = cell_length * cell_length;
113
114 for (i = 0; i < nbr_atoms; i++){
115     sx[i] = positions[i][0] * cell_length_inv;
116     sy[i] = positions[i][1] * cell_length_inv;
117     sz[i] = positions[i][2] * cell_length_inv;
118 }
119
120 for (i = 0; i < nbr_atoms; i++){
121     density[i] = 0;
122     fx[i] = 0;
123     fy[i] = 0;
124     fz[i] = 0;
125 }
126
127 for (i = 0; i < nbr_atoms; i++) {
128     /* Periodically translate coords of current particle to positive quadrants ←
129     */
130     sxi = sx[i] - floor(sx[i]);
131     syi = sy[i] - floor(sy[i]);
132     szi = sz[i] - floor(sz[i]);
133
134     densityi = density[i];
135
136     /* Loop over other atoms. */
137     for (j = i + 1; j < nbr_atoms; j++) {
138         /* Periodically translate atom j to positive quadrants and calculate ←
139         distance to it. */
140         sxij = sxi - (sx[j] - floor(sx[j]));
141         syij = syi - (sy[j] - floor(sy[j]));
142         szij = szi - (sz[j] - floor(sz[j]));
143
144         /* Periodic boundary conditions. */
145         sxij = sxij - (int)floor(sxij + 0.5);
146         syij = syij - (int)floor(syij + 0.5);
147         szij = szij - (int)floor(szij + 0.5);
148
149         /* squared distance between atom i and j */
150         rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
151
152         /* Add force and energy contribution if distance between atoms smaller ←
153         than rcut */
154         if (rij_sq < rcut_sq) {
155             rij = sqrt(rij_sq);
156             dens = splineEval(rij, electron_density, ELECTRON_DENSITY_ROWS);
157             densityi += dens;
158             density[j] += dens;
159         }
160     }
161     density[i] = densityi;
162 }
163
164 /* Loop over atoms to calculate derivative of embedding function
165 and embedding function. */
166 for (i = 0; i < nbr_atoms; i++) {
167     dUembed_drho[i] = splineEvalDiff(density[i], embedding_energy, ←
168     EMBEDDING_ENERGY_ROWS);
169 }
170
171 /* Compute forces on atoms. */
172 /* Loop over atoms again :-(. */
173
174 for (i = 0; i < nbr_atoms; i++) {
175     /* Periodically translate coords of current particle to positive quadrants ←
176     */
177     sxi = sx[i] - floor(sx[i]);
178     syi = sy[i] - floor(sy[i]);
179     szi = sz[i] - floor(sz[i]);
180
181     densityi = density[i];
182
183     /* Loop over other atoms. */
184     for (j = i + 1; j < nbr_atoms; j++) {
185         /* Periodically translate atom j to positive quadrants and calculate ←
186         distance to it. */
187         sxij = sxi - (sx[j] - floor(sx[j]));
188         syij = syi - (sy[j] - floor(sy[j]));
189         szij = szi - (sz[j] - floor(sz[j]));

```



```

184
185 /* Periodic boundary conditions. */
186     sxij = sxij - (int)floor(sxij + 0.5);
187     syij = syij - (int)floor(syij + 0.5);
188     szij = szij - (int)floor(szij + 0.5);
189
190 /* squared distance between atom i and j */
191     rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
192
193 /* Add force and energy contribution if distance between atoms smaller ←
    than rcut */
194     if (rij_sq < rcut_sq) {
195         rij = sqrt(rij_sq);
196         dUpair_dr = splineEvalDiff(rij, pair_potential, PAIR_POTENTIAL_ROWS);
197         drho_dr = splineEvalDiff(rij, electron_density, ELECTRON_DENSITY_ROWS);
198
199 /* Add force contribution from i-j interaction */
200         force = -(dUpair_dr + (dUembed_drho[i] + dUembed_drho[j])←
            drho_dr) / rij;
201         fx[i] += force * sxij * cell_length;
202         fy[i] += force * syij * cell_length;
203         fz[i] += force * szij * cell_length;
204         fx[j] -= force * sxij * cell_length;
205         fy[j] -= force * syij * cell_length;
206         fz[j] -= force * szij * cell_length;
207     }
208 }
209 }
210
211 for (i = 0; i < nbr_atoms; i++){
212     forces[i][0] = fx[i];
213     forces[i][1] = fy[i];
214     forces[i][2] = fz[i];
215 }
216
217 free(sx); free(sy); free(sz); sx = NULL; sy = NULL; sz = NULL;
218 free(fx); free(fy); free(fz); fx = NULL; fy = NULL; fz = NULL;
219 free(density); density = NULL;
220 free(dUembed_drho); dUembed_drho = NULL;
221 }
222 }
223
224 /* Returns the potential energy */
225 double get_energy_AL(double positions[][3], double cell_length, int nbr_atoms)
226 {
227     int i, j;
228     double cell_length_inv, cell_length_sq;
229     double rcut, rcut_sq;
230     double energy;
231     double densityi, dens;
232     double sxi, syi, szi, sxij, syij, szij, rij, rij_sq;
233
234     double *sx = malloc(nbr_atoms * sizeof (double));
235     double *sy = malloc(nbr_atoms * sizeof (double));
236     double *sz = malloc(nbr_atoms * sizeof (double));
237
238     double *density = malloc(nbr_atoms * sizeof (double));
239
240     rcut = 6.06;
241     rcut_sq = rcut * rcut;
242
243     cell_length_inv = 1 / cell_length;
244     cell_length_sq = cell_length * cell_length;
245
246     for (i = 0; i < nbr_atoms; i++){
247         sx[i] = positions[i][0] * cell_length_inv;
248         sy[i] = positions[i][1] * cell_length_inv;
249         sz[i] = positions[i][2] * cell_length_inv;
250     }
251
252     for (i = 0; i < nbr_atoms; i++){
253         density[i] = 0;
254     }
255
256     energy = 0;
257
258     for (i = 0; i < nbr_atoms; i++) {
259         /* Periodically translate coords of current particle to positive quadrants ←
            */
260         sxi = sx[i] - floor(sx[i]);
261         syi = sy[i] - floor(sy[i]);
262         szi = sz[i] - floor(sz[i]);
263
264         densityi = density[i];
265
266         /* Loop over other atoms. */
267         for (j = i + 1; j < nbr_atoms; j++) {
268             /* Periodically translate atom j to positive quadrants and calculate ←
                distance to it. */
269             sxij = sxi - (sx[j] - floor(sx[j]));
270             syij = syi - (sy[j] - floor(sy[j]));

```

```

271         szij = szi - (sz[j] - floor(sz[j]));
272
273     /* Periodic boundary conditions. */
274     sxij = sxij - (int)floor(sxij + 0.5);
275     syij = syij - (int)floor(syij + 0.5);
276     szij = szij - (int)floor(szij + 0.5);
277
278     /* squared distance between atom i and j */
279     rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
280
281     /* Add force and energy contribution if distance between atoms smaller <-
        than rcut */
282     if (rij_sq < rcut_sq) {
283         rij = sqrt(rij_sq);
284         dens = splineEval(rij, electron_density, ELECTRON_DENSITY_ROWS);
285         densityi += dens;
286         density[j] += dens;
287
288         /* Add energy contribution from i-j interaction */
289         energy += splineEval(rij, pair_potential, PAIR_POTENTIAL_ROWS);
290     }
291 }
292 density[i] = densityi;
293 }
294 }
295
296 /* Loop over atoms to calculate derivative of embedding function
297 and embedding function. */
298 for (i = 0; i < nbr_atoms; i++) {
299     energy += splineEval(density[i], embedding_energy, EMBEDDING_ENERGY_ROWS);
300 }
301
302 free(sx); free(sy); free(sz); sx = NULL; sy = NULL; sz = NULL;
303 free(density); density = NULL;
304
305 return(energy);
306
307 }
308
309 /* Returns the virial */
310 double get_virial_AL(double positions[][3], double cell_length, int nbr_atoms)
311 {
312     int i, j;
313     double cell_length_inv, cell_length_sq;
314     double rcut, rcut_sq;
315     double virial;
316     double densityi, dens, drho_dr, force;
317     double dUpair_dr;
318     double sxi, syi, szi, sxij, syij, szij, rij, rij_sq;
319
320     double *sx = malloc(nbr_atoms * sizeof (double));
321     double *sy = malloc(nbr_atoms * sizeof (double));
322     double *sz = malloc(nbr_atoms * sizeof (double));
323
324     double *density = malloc(nbr_atoms * sizeof (double));
325     double *dUembed_drho = malloc(nbr_atoms * sizeof (double));
326
327     rcut = 6.06;
328     rcut_sq = rcut * rcut;
329
330     cell_length_inv = 1 / cell_length;
331     cell_length_sq = cell_length * cell_length;
332
333     for (i = 0; i < nbr_atoms; i++){
334         sx[i] = positions[i][0] * cell_length_inv;
335         sy[i] = positions[i][1] * cell_length_inv;
336         sz[i] = positions[i][2] * cell_length_inv;
337     }
338
339     for (i = 0; i < nbr_atoms; i++){
340         density[i] = 0;
341     }
342
343     for (i = 0; i < nbr_atoms; i++) {
344         /* Periodically translate coords of current particle to positive quadrants <-
            */
345         sxi = sx[i] - floor(sx[i]);
346         syi = sy[i] - floor(sy[i]);
347         szi = sz[i] - floor(sz[i]);
348
349         densityi = density[i];
350
351         /* Loop over other atoms. */
352         for (j = i + 1; j < nbr_atoms; j++) {
353             /* Periodically translate atom j to positive quadrants and calculate <-
                distance to it. */
354             sxij = sxi - (sx[j] - floor(sx[j]));
355             syij = syi - (sy[j] - floor(sy[j]));
356             szij = szi - (sz[j] - floor(sz[j]));
357

```

```

358     /* Periodic boundary conditions. */
359     sxij = sxij - (int)floor(sxij + 0.5);
360     syij = syij - (int)floor(syij + 0.5);
361     szij = szij - (int)floor(szij + 0.5);
362
363     /* squared distance between atom i and j */
364     rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
365
366     /* Add force and energy contribution if distance between atoms smaller ←
        than rcut */
367     if (rij_sq < rcut_sq) {
368         rij = sqrt(rij_sq);
369         dens = splineEval(rij, electron_density, ELECTRON_DENSITY_ROWS);
370         densityi += dens;
371         density[j] += dens;
372     }
373 }
374 density[i] = densityi;
375 }
376
377 /* Loop over atoms to calculate derivative of embedding function
378 and embedding function. */
379 for (i = 0; i < nbr_atoms; i++) {
380     dUembed_drho[i] = splineEvalDiff(density[i], embedding_energy, ←
        EMBEDDING_ENERGY_ROWS);
381 }
382
383 /* Compute forces on atoms. */
384 /* Loop over atoms again :-(. */
385
386 virial = 0;
387
388 for (i = 0; i < nbr_atoms; i++) {
389     /* Periodically translate coords of current particle to positive quadrants ←
        */
390     sxi = sx[i] - floor(sx[i]);
391     syi = sy[i] - floor(sy[i]);
392     szi = sz[i] - floor(sz[i]);
393
394     densityi = density[i];
395
396     /* Loop over other atoms. */
397     for (j = i + 1; j < nbr_atoms; j++) {
398         /* Periodically translate atom j to positive quadrants and calculate ←
            distance to it. */
399         sxij = sxi - (sx[j] - floor(sx[j]));
400         syij = syi - (sy[j] - floor(sy[j]));
401         szij = szi - (sz[j] - floor(sz[j]));
402
403         /* Periodic boundary conditions. */
404         sxij = sxij - (int)floor(sxij + 0.5);
405         syij = syij - (int)floor(syij + 0.5);
406         szij = szij - (int)floor(szij + 0.5);
407
408         /* squared distance between atom i and j */
409         rij_sq = cell_length_sq * (sxij*sxij + syij*syij + szij*szij);
410
411         /* Add force and energy contribution if distance between atoms smaller ←
            than rcut */
412         if (rij_sq < rcut_sq) {
413             rij = sqrt(rij_sq);
414             dUpair_dr = splineEvalDiff(rij, pair_potential, PAIR_POTENTIAL_ROWS);
415             drho_dr = splineEvalDiff(rij, electron_density, ←
                ELECTRON_DENSITY_ROWS);
416
417             /* Add virial contribution from i-j interaction */
418             force = -(dUpair_dr + (dUembed_drho[i] + dUembed_drho[j])*←
                drho_dr) / rij;
419
420             virial += force * rij_sq;
421         }
422     }
423 }
424
425 virial /= 3.0;
426
427 free(sx); free(sy); free(sz); sx = NULL; sy = NULL; sz = NULL;
428 free(density); density = NULL;
429 free(dUembed_drho); dUembed_drho = NULL;
430
431 return(virial);
432 }
433

```

A.5 FCC initialization file initfcc.c

```

1  /*
2  initfcc.c
3  Program that arranges atoms on a fcc lattice.
4  Created by Anders Lindman on 2013-03-15.
5  */
6
7  #include <stdio.h>
8
9  /* Function takes a matrix of size [4*N*N*N][3] as input and stores a fcc ↵
10     lattice in it. N is the number of unit cells in each dimension and ↵
11     lattice_param is the lattice parameter. */
12 void init_fcc(double positions[][3], int N, double lattice_param)
13 {
14     int i, j, k;
15     int xor_value;
16
17     for (i = 0; i < 2 * N; i++){
18         for (j = 0; j < 2 * N; j++){
19             for (k = 0; k < N; k++){
20                 if (j % 2 == i % 2){
21                     xor_value = 0;
22                 }
23                 else {
24                     xor_value = 1;
25                 }
26                 positions[i * N * 2 * N + j * N + k][0] = lattice_param * (0.5 * ↵
27                     xor_value + k);
28                 positions[i * N * 2 * N + j * N + k][1] = lattice_param * (j * ↵
29                     0.5);
30                 positions[i * N * 2 * N + j * N + k][2] = lattice_param * (i * ↵
31                     0.5);
32             }
33         }
34     }
35 }

```

A.6 plot_task1.py

```

1  import numpy as np
2  import matplotlib as mpl
3  from matplotlib import pyplot as plt
4  import struct
5
6  ### for plotting nice plots ###
7  width = 6
8  height = width/1.5
9  fsize = 16
10 font = {'size': fsize}
11 mpl.rc('font', **font)
12 mpl.rc('xtick', labelsizes=fsize)
13 mpl.rc('ytick', labelsizes=fsize)
14 mpl.rc('text', usetex=False)
15
16 ### read files generated by C ###
17 potential_energy = np.fromfile('potential_energy.txt', sep='\n')
18 a0 = np.fromfile('a0.txt', sep='\n')
19
20 ### plot E(a) ###
21 fig, ax = plt.subplots()
22 ax.plot(a0**3, potential_energy, '.')
23 ax.set_xlabel(r'$a^3$ [Å3])')
24 ax.set_ylabel(r'$E_{\mathrm{p}}$ [eV / unit cell]')
25
26 #plt.legend()
27 plt.tight_layout()
28 filename = f'plot_task1.pdf'
29 plt.savefig(filename)
30 plt.show()

```

A.7 plot_task2.py

```

1  import numpy as np
2  import matplotlib as mpl
3  from matplotlib import pyplot as plt
4  import struct
5
6  ### for plotting nice plots ###
7  width = 6
8  height = width / 1.5
9  fsize = 16
10 font = {'size': fsize}

```

```

11 mpl.rc('font', **font)
12 mpl.rc('xtick', labelsizes=fsize)
13 mpl.rc('ytick', labelsizes=fsize)
14 mpl.rc('text', usetex=False)
15
16 ### read binary files generated by C ###
17 potential_energies = np.fromfile('potential_energies_task2.bin')
18 kinetic_energies = np.fromfile('kinetic_energies_task2.bin')
19 total_energies = potential_energies + kinetic_energies
20 parameters = np.fromfile('parameters_task2.bin')
21
22 ### extract parameters used ###
23 nbr_timesteps = int(parameters[0])
24 dt = parameters[1]
25 nbr_atoms = parameters[2]
26 nbr_cells = nbr_atoms / 4
27 a0 = parameters[3]
28 t = np.array([i * dt for i in range(0, nbr_timesteps)])
29
30 ### prepare broken axis ###
31 fig = plt.figure(figsize=(8,5))
32 ax = fig.add_subplot(111) # The big subplot
33 ax1 = fig.add_subplot(211)
34 ax2 = fig.add_subplot(212)
35
36 # Turn off axis lines and ticks of the big subplot
37 ax.spines['top'].set_color('none')
38 ax.spines['bottom'].set_color('none')
39 ax.spines['left'].set_color('none')
40 ax.spines['right'].set_color('none')
41 ax.set_xticks([])
42 ax.set_yticks([])
43
44 # do some other stuff
45 ax1.spines['bottom'].set_visible(False)
46 ax2.spines['top'].set_visible(False)
47 ax1.set_xticks([])
48 ax2.xaxis.tick_bottom()
49 d = .015 # how big to make the diagonal lines in axes coordinates
50
51 # arguments to pass to plot, just so we don't keep repeating them
52 kwargs = dict(transform=ax1.transAxes, color='k', clip_on=False)
53 ax1.plot((-d, +d), (-d, +d), **kwargs) # top-left diagonal
54 ax1.plot((1 - d, 1 + d), (-d, +d), **kwargs) # top-right diagonal
55 kwargs.update(transform=ax2.transAxes) # switch to the bottom axes
56 ax2.plot((-d, +d), (1 - d, 1 + d), **kwargs) # bottom-left diagonal
57 ax2.plot((1 - d, 1 + d), (1 - d, 1 + d), **kwargs) # bottom-right diagonal
58
59 ### plot E(t) ###
60 l0 = ax2.plot(t, potential_energies / nbr_cells, label=r'potential energy', ←
61               color='C0')
62 l1 = ax1.plot(t, kinetic_energies / nbr_cells, label=r'kinetic energy', color='←
63               C1')
64 l2 = ax2.plot(t, total_energies / nbr_cells, label=r'total energy', color='C2')
65 ax.set_xlabel('\n' + r'$t$ [ps]')
66 ax.set_ylabel(r'$E$ [eV / unit cell]' + '\n\n\n')
67 ax1.set_ylim(top=0.7, bottom=-0.3)
68 ax1.set_yticks([0.6, 0.4, 0.2, 0.0])
69 ax2.set_yticks([-12.8, -13.0, -13.2])
70 ax2.set_ylim(top=-12.3, bottom=-13.3)
71
72 ### plot 800 K line ###
73 kB = 8.617e-5 # Boltzmann constant [eV / K]
74 E_800K = (3 / 2) * kB * 800 * 4 # *4 since 4 atoms per unit cell
75 l3 = ax1.plot([min(t), max(t)], [E_800K, E_800K], 'r--', zorder=-1, label=r'←
76               kinetic energy at 800 K')
77
78 ### legend & save file ###
79 fig.legend((l0[0], l1[0], l2[0], l3[0]),
80            (r'potential energy', r'kinetic energy', r'total energy', r'kinetic ←
81             energy at 800 K'),
82            loc=(0.38, 0.48))
83 plt.tight_layout()
84 filename = f'plot_task2_dt=' + str(dt) + '.pdf'
85 plt.savefig(filename, bbox_inches='tight')
86 plt.show()

```

A.8 plot_task3 and 4 .py

```

1 import numpy as np
2 import matplotlib as mpl
3 from matplotlib import pyplot as plt
4 import struct
5
6 ### for plotting nice plots ###

```

```

7 width = 6
8 height = width / 1.5
9 fsize = 16
10 font = {'size': fsize}
11 mpl.rc('font', **font)
12 mpl.rc('xtick', labelsizes=fsize)
13 mpl.rc('ytick', labelsizes=fsize)
14 mpl.rc('text', usetex=False)
15
16 ### read binary files generated by C ###
17 temperatures_solid = np.fromfile('temperatures_task3.bin')
18 pressures_solid = np.fromfile('pressures_task3.bin')
19 parameters_solid = np.fromfile('parameters_task3.bin')
20 temperatures_liquid = np.fromfile('temperatures_task4.bin')
21 pressures_liquid = np.fromfile('pressures_task4.bin')
22 parameters_liquid = np.fromfile('parameters_task4.bin')
23 indices_solid = np.array([i for i in range(len(temperatures_solid))])
24 indices_liquid = np.array([i for i in range(len(temperatures_liquid))])
25
26 ### extract parameters ###
27 nbr_timesteps_per_epoch = int(parameters_solid[0])
28 nbr_epochs = int(parameters_solid[1])
29 T_decay_constant = parameters_solid[2]
30 P_decay_constant = parameters_solid[3]
31 timestep = parameters_solid[5]
32
33 ### plot T(t) & P(t) ###
34 fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(8, 8))
35 ax1.plot(timestep * indices_solid, temperatures_solid, label=r'solid phase', ←
        color='C0')
36 ax1.plot(timestep * indices_liquid, temperatures_liquid, label=r'liquid phase', ←
        color='C1')
37 ax1.set_xlabel(r'$t$ [ps]')
38 ax1.set_ylabel(r'$T$ [K]')
39 ax2.plot(timestep * indices_solid, pressures_solid, label=r'solid phase', color=←
        'C0')
40 ax2.plot(timestep * indices_liquid, pressures_liquid, label=r'liquid phase', ←
        color='C1')
41 ax2.set_xlabel(r'$t$ [ps]')
42 ax2.set_ylabel(r'$P$ [eV / Å$^3$]')
43
44 ### plot 500 C line ###
45 T_eq_solid = 500 + 273.15
46 P_eq_solid = 6.242e-7
47 T_eq_liquid = 700 + 273.15
48 P_eq_liquid = 6.242e-7
49 ax1.plot(timestep * indices_solid, T_eq_solid * np.ones(indices_solid.shape), '←
        --', zorder=-1, label=r'500 $\mathrm{o}^\circ\mathrm{C}$', color='k')
50 ax1.plot(timestep * indices_liquid, T_eq_liquid * np.ones(indices_liquid.shape), ←
        ':', zorder=-1, label=r'700 $\mathrm{o}^\circ\mathrm{C}$', color='k')
51 ax2.plot(timestep * indices_solid, P_eq_solid * np.ones(indices_solid.shape), '←
        --', zorder=-1, label=r'1 bar', color='k')
52
53 ### legend & save file ###
54 ax1.legend(loc=1)
55 ax2.legend()
56 plt.tight_layout()
57 filename = f'plot_task3_and_4.pdf'
58 plt.savefig(filename)
59 plt.show()

```

A.9 plot_task5.py

```

1 import numpy as np
2 import matplotlib as mpl
3 from matplotlib import pyplot as plt
4 import struct
5
6 ### for plotting nice plots ###
7 width = 6
8 height = width / 1.5
9 fsize = 16
10 font = {'size': fsize}
11 mpl.rc('font', **font)
12 mpl.rc('xtick', labelsizes=fsize)
13 mpl.rc('ytick', labelsizes=fsize)
14 mpl.rc('text', usetex=False)
15
16 ### read binary files generated by C ###
17 mean_squared_displacements = np.fromfile('mean_squared_displacements_task5.bin')
18 parameters = np.fromfile('parameters_task5.bin')
19
20 ### extract parameters ###
21 dt = parameters[0]
22 nbr_timesteps = int(parameters[1])
23 t = np.array([i * dt for i in range(nbr_timesteps)])

```

```

24
25 ### plot mean_squared_displacements(t) ###
26 fig, ax = plt.subplots()
27 ax.plot(t[1:-1], mean_squared_displacements[1:-1], label=r'mean squared ↵
    displacement')
28 ax.set_xlabel(r'$t$ [ps]')
29 ax.set_ylabel(r'$A$')
30
31 ### legend & save file ###
32 plt.legend()
33 plt.tight_layout()
34 filename = f'plot_task5_timesteps={nbr_timesteps}_dt={dt}.pdf'
35 plt.savefig(filename)
36 plt.show()

```

A.10 plot_task6.py

```

1 import numpy as np
2 import matplotlib as mpl
3 from matplotlib import pyplot as plt
4 import struct
5
6 ### for plotting nice plots ###
7 width = 6
8 height = width / 1.5
9 fsize = 16
10 font = {'size': fsize}
11 mpl.rc('font', **font)
12 mpl.rc('xtick', labelsz=fsize)
13 mpl.rc('ytick', labelsz=fsize)
14 mpl.rc('text', usetex=False)
15
16 ### read binary files generated by C ###
17 velocity_correlations = np.fromfile('velocity_correlations_task6.bin')
18 parameters = np.fromfile('parameters_task6.bin')
19
20 ### extract parameters ###
21 dt = parameters[0]
22 nbr_timesteps = int(parameters[1])
23 t = np.array([i * dt for i in range(nbr_timesteps)])
24
25 ### plot velocity_correlations(t) ###
26 fig, ax = plt.subplots()
27 ax.plot(t, velocity_correlations, label=r'veLOCITY correlation')
28 ax.set_xlabel(r'$t$ [ps]')
29 ax.set_ylabel(r'$A$^2$ / $ps$^2$')
30
31 ### legend & save file ###
32 plt.legend()
33 plt.tight_layout()
34 filename = f'plot_task6.pdf'
35 plt.savefig(filename)
36 plt.show()
37
38 ### calculate cosine transform ###
39 D = np.trapz(velocity_correlations, x=t) / 3
40 print(f'D = {D} []')

```

A.11 plot_task6.py

```

1 import numpy as np
2 import matplotlib as mpl
3 from matplotlib import pyplot as plt
4 import struct
5
6 ### for plotting nice plots ###
7 width = 6
8 height = width / 1.5
9 fsize = 16
10 font = {'size': fsize}
11 mpl.rc('font', **font)
12 mpl.rc('xtick', labelsz=fsize)
13 mpl.rc('ytick', labelsz=fsize)
14 mpl.rc('text', usetex=False)
15
16 ### read binary files generated by C ###
17 powerspectrum = np.fromfile('powerspectrum_task7.bin')
18 frequencies = np.fromfile('frequencies_task7.bin')
19 correlation = 1j * np.zeros(powerspectrum.shape)
20 N = len(powerspectrum)
21 for l in range(N):

```

```

22     for n in range(N - 1):
23         correlation[1] += powerspectrum[n]**2 * np.exp(2 * np.pi * 1j * l * n / ←
           N) / N
24
25     ### plot velocity_correlations(t) ###
26     fig, ax = plt.subplots()
27     ax.plot(1e-3 * np.array([i for i in range(N)]), np.real(correlation), label=r'←
           velocity correlation')
28     ax.set_xlabel(r'[ps$^{-1}$]')
29     ax.set_ylabel(r'[A$^2$ / ps$^2$]')
30     ax.set_xlim(left=0.0, right=1e-3 * N / 2)
31     ax.set_yticks([])
32
33     ### legend & save file ###
34     plt.legend()
35     plt.tight_layout()
36     filename = f'plot_task7.pdf'
37     plt.savefig(filename)
38     plt.show()

```