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 №	Points	Avail. points
$\sum$		

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 (ffc) structure with periodic boundary conditions, i.e. an infinite system.

Task 1

The Lennard-Jones potential for aluminium in an fcc structrue 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 $\sqrt[3]{65.5} \approx 4.03$.

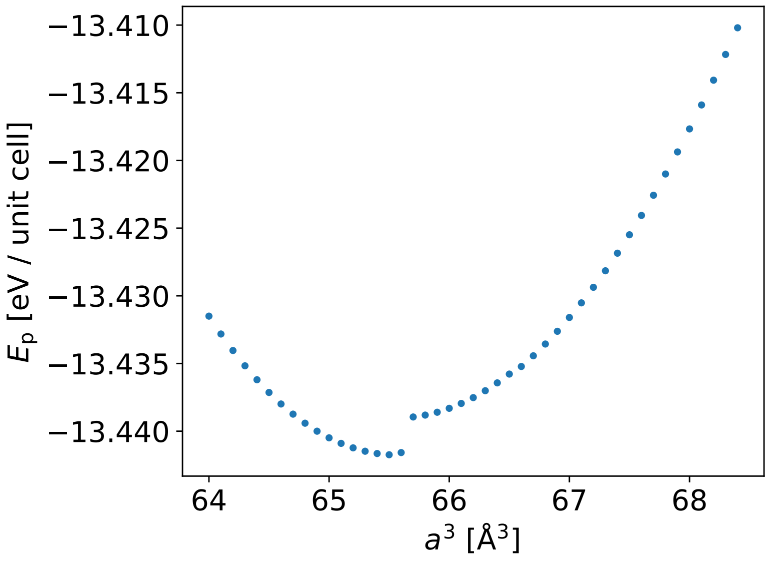

Figure 1: Shows potential energy as a function of unit-cell volume for aluminium. That is the minimum gives the lattice constant at 0 K, $\sqrt[3]{65.5} \approx 4.03$

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 ±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,

$$\begin{cases} \mathbf{v}(t+\frac{\Delta t}{2}) = \mathbf{v}(t) + \frac{1}{2}\mathbf{a}(t)\Delta t \\ \mathbf{r}(t+\Delta t) = \mathbf{r}(t) + \mathbf{v}(t+\frac{\Delta t}{2})\Delta t \\ \text{Update accelerations by } \texttt{get\_forces\_AL()} \\ \mathbf{v}(t) + \frac{1}{2}\mathbf{a}(t+\Delta t)\Delta t \end{cases} \tag{1}$$

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 us 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.

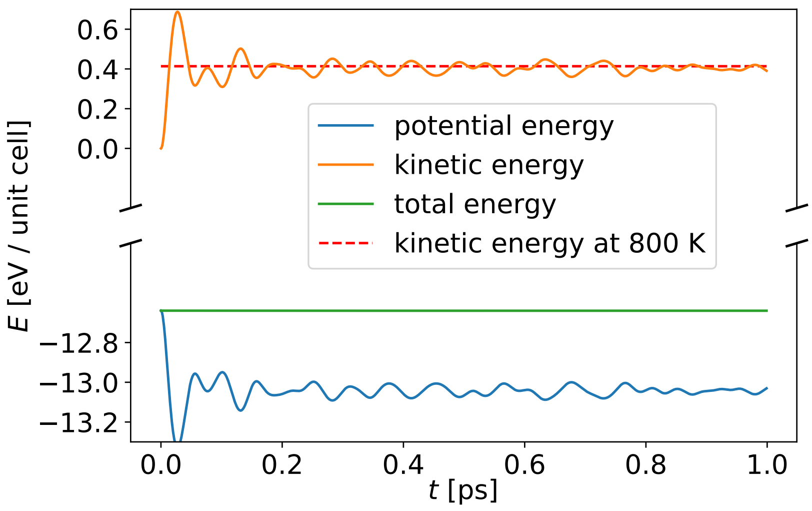

Figure 2: The time evolution of the kinetic, potential, and total energy per unit cell, calculated with time step 0.001 ps. After about 0.2 ps, the oscillations between kinetic and potential energy appear to have stabilized. The total energy is also clearly conserved throughout the process.

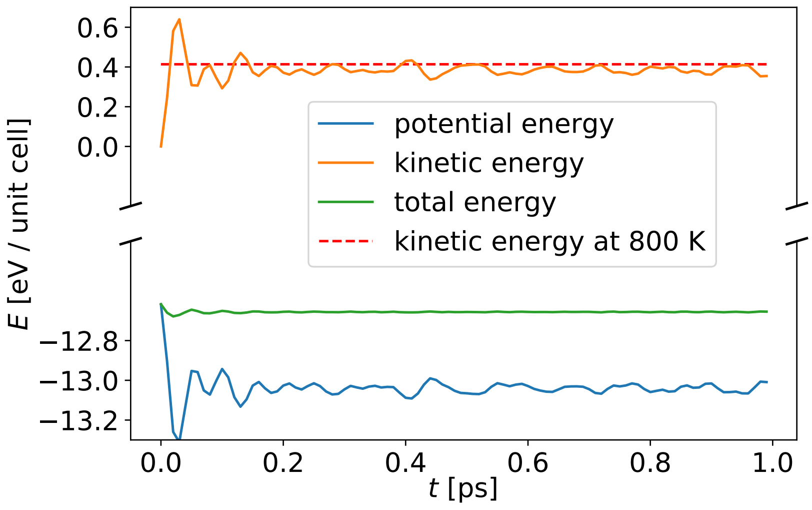

Figure 3: The kinetic, potential, and total energy per unit cell, calculated with time step 0.01 ps. As in Figure 2, there appear to be larger oscillations between kinetic and potential energy within 0.2 ps of start. Unlike in Figure 2, the total energy seems to oscillate around it's average value.

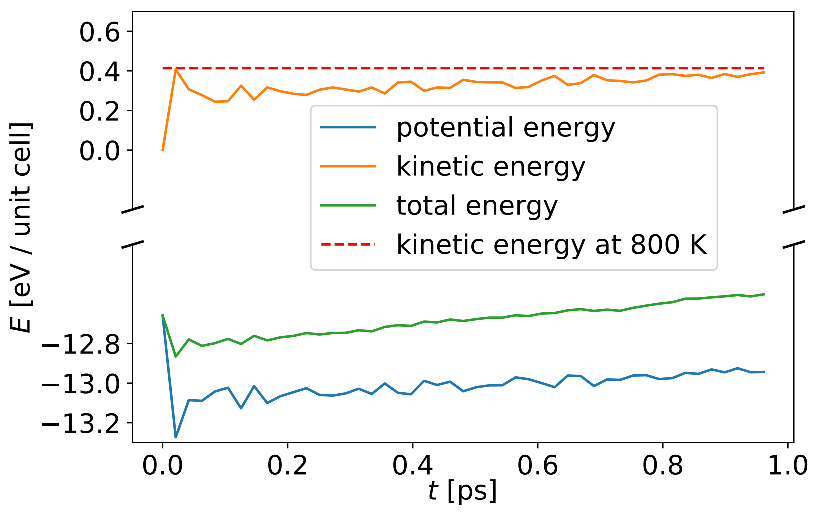

Figure 4: The kinetic, potential, and total energy per unit cell, calculated with time step ~0.021 ps. The total energy is clearly not conserved throughout the process, implying that this is a too large time step.

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 equlibration, 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 equlibration 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).

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

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

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):

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

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

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 Å^3/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, sufficently long to have the system with the specified temprature and pressure we go in to production period where we simply turn off the rescaling of postion 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 ±75 K. But in the production, where we have turned of the scaling used in the equlibration 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.

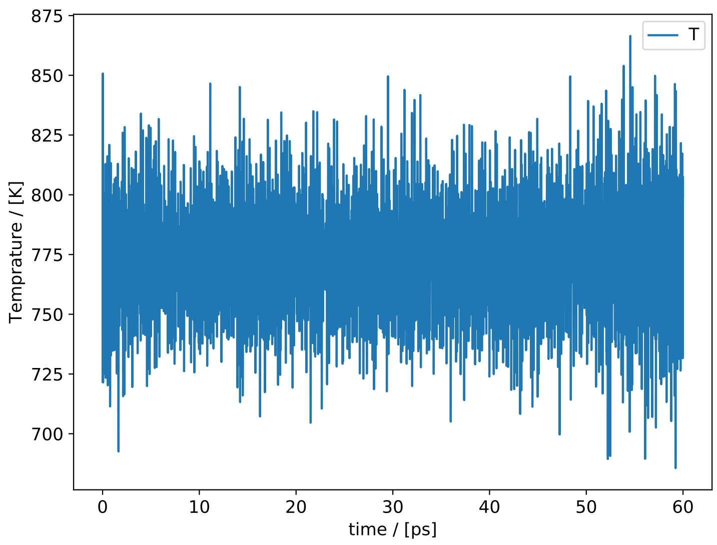

Figure 5: Temperature time evolution for task 3.

The pressure has bigger oscillations around ±0.001 eV/Å^3, which is around ±0.16 GPa. In the production from 50 ps to 60 ps where we average the pressure we get a value of 0.000 056 eV/Å^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 reach 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 Å.

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 equlibrate 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

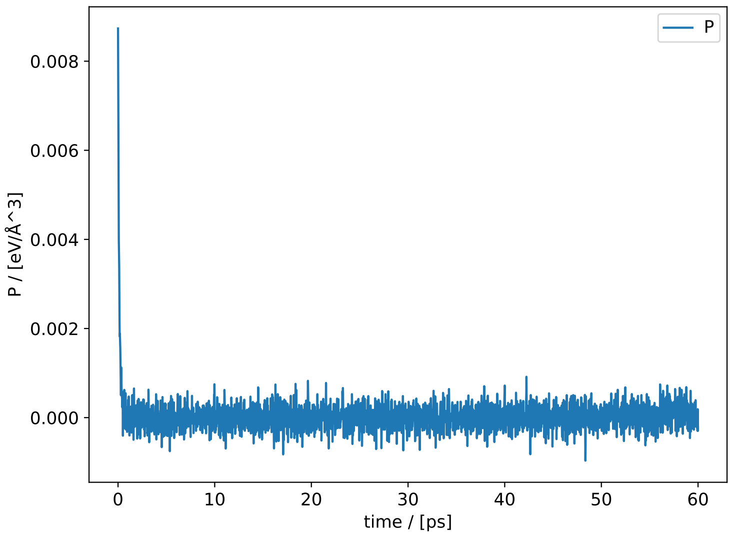

Figure 6: Pressure time evolution for task 3.

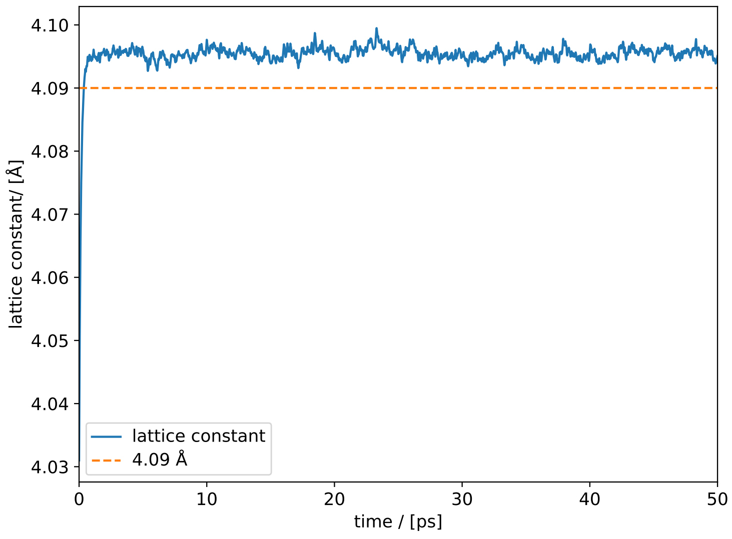

Figure 7: Lattice constant time evolution for task 3.

it is preferable to have melting period which is exactly the same as the equlibration 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 temprature, ±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 temprature 973 K.

The pressure got bigger oscillations than the temprature, of around $\pm 0.0015\,\mathrm{eV}/\text{Å}^3$ which is in ”ordinary” units ±0.24 GPa. But in the production run the last 10 ps we average the pressure and get $0.000\,035\,\mathrm{eV}/\text{Å}^3$, 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 equlibrium. 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 equlibration period which is the way to change pressure and thus a stabilized lattice constant is an indicator for reaching equilibrium.

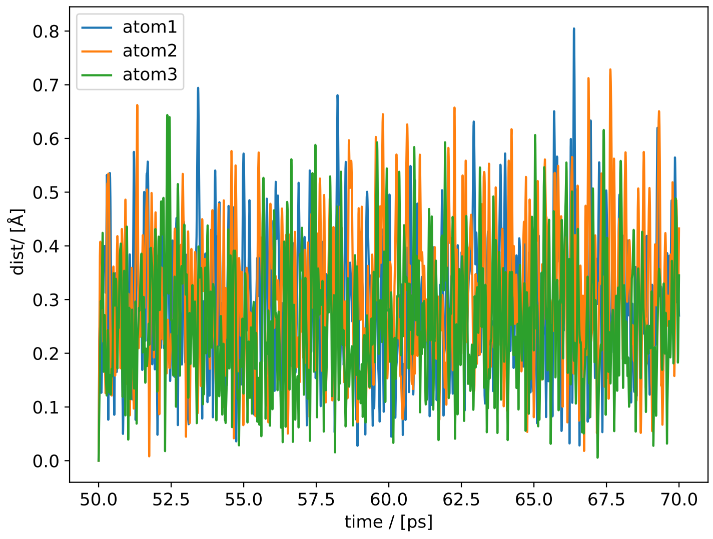

Figure 8: Distance for three atoms, atom 1, 100, & 200, to there respectively first position in the production run. We can see that they oscillate around that position, i.e. no drift so we have solid phase.

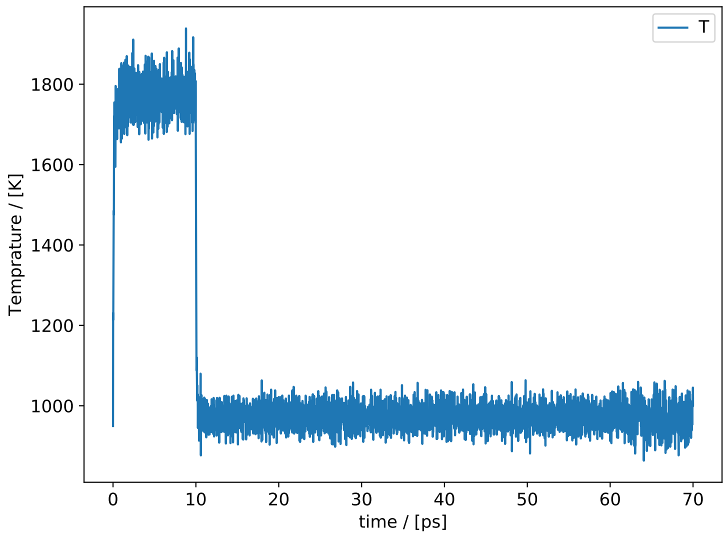

Figure 9: Temperature time evolution for task 4.

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.

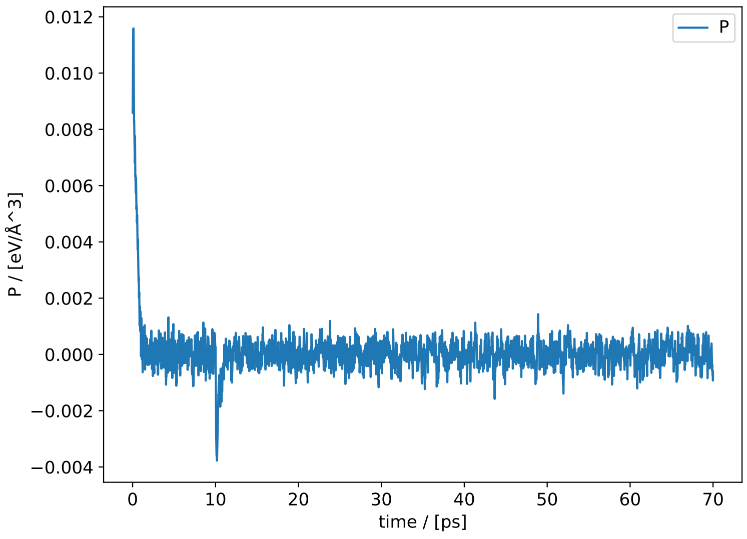

Figure 10: Pressure time evolution for task 4.

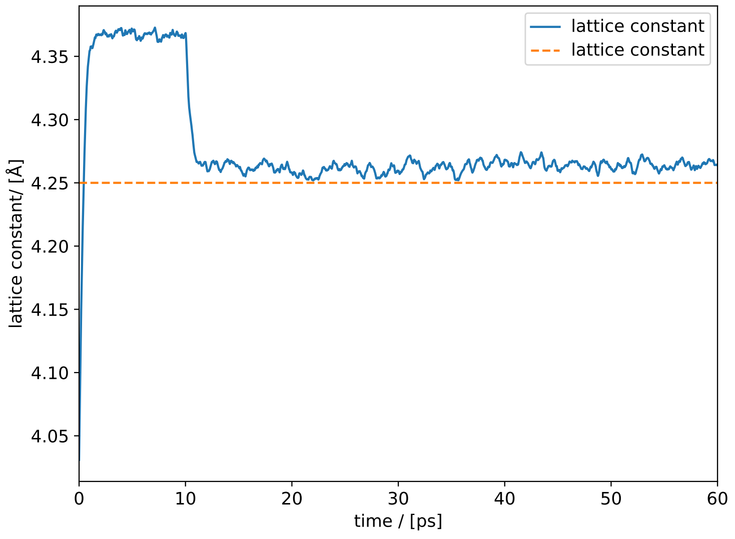

Figure 11: Temperature time evolutions for task 4.

Task 5

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

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

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

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

After an identical equlibration 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 equlibration is the same as

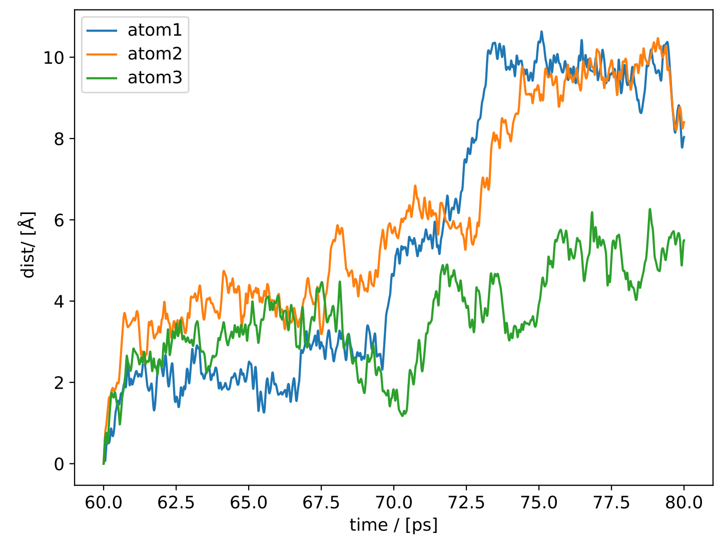

Figure 12: Temperature time evolution for task 4. The reason for keeping the sample at a much higher temperature fro the first 10 ps is that we want to be sure that the sample is in a liquid state.

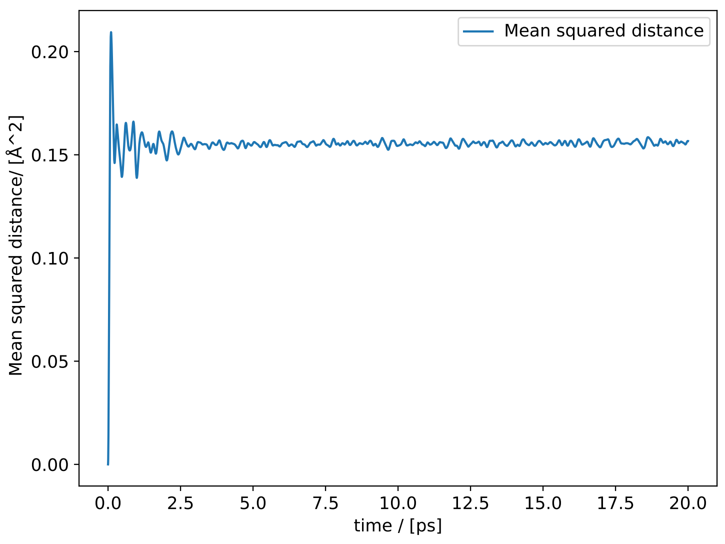

Figure 13: Mean squared displacements of the particles for the solid phase.

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

Fick's law,

$$\mathbf{j} = -D\nabla n, \tag{7}$$

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

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

into (6), we get

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

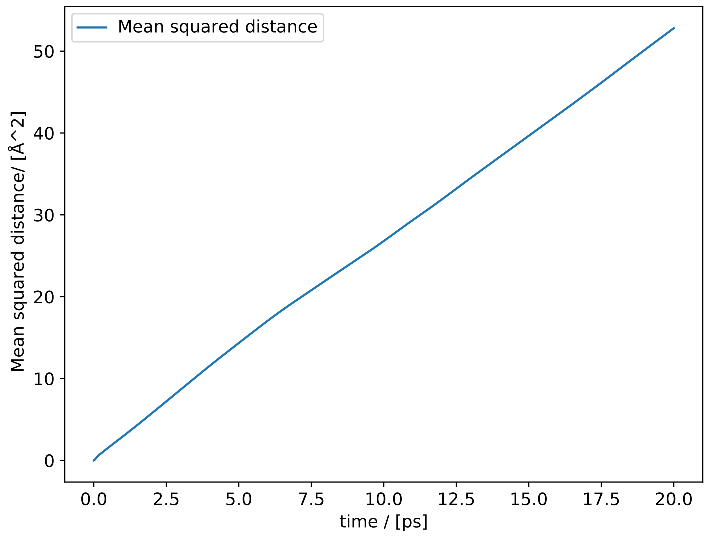

Figure 14: Mean squared displacements of the particles for the liquid phase. Taking the last point on the plot (20, 52.8048) giving the self diffusion coefficient to be 0.44 Å^2/ps

This is a heat equation with kernel

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

Since (6) is a space average,

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

$$= \int_0^\infty \int_0^t r^2 \frac{1}{(4\pi Dt')^{\frac{3}{2}}} \exp\left[\frac{-r^2}{4Dt'}\right] r^2 4\pi \mathrm{d}r\mathrm{d}t' \xrightarrow{t\to\infty} 6Dt. \tag{12}$$

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

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

is obtained.

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

Task 6

The *velocity correlation* is defined as

$$\Phi(t) = \left\langle\left\langle \mathbf{v}_i(t + t') \cdot \mathbf{v}_i(t')\right\rangle_i\right\rangle_{t'}. \tag{14}$$

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

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

where

$$\hat{\Phi}(\omega) = 2\int_0^\infty \Phi(t)\cos\omega t \,\mathrm{d}t. \tag{16}$$

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

$$D = \frac{1}{3}\int_0^\infty \Phi(t)\,\mathrm{d}t. \tag{17}$$

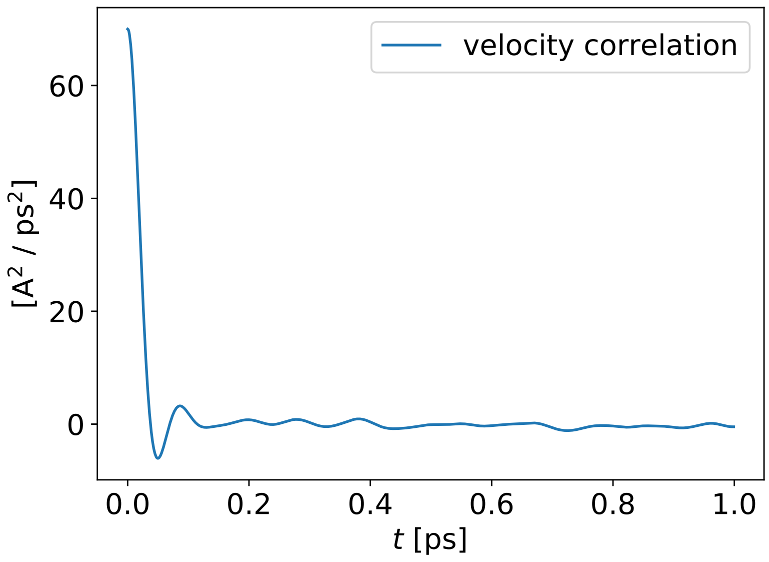

Figure 15: The velocity correlation plotted as a function of time for a liquid aluminum sample at 700 °C.

From this we obtain $D \approx 0.4$ Å. 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

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

where

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

and

$$h_k = h_{k+2N}. \tag{20}$$

From the circular correlation theorem,

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

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).

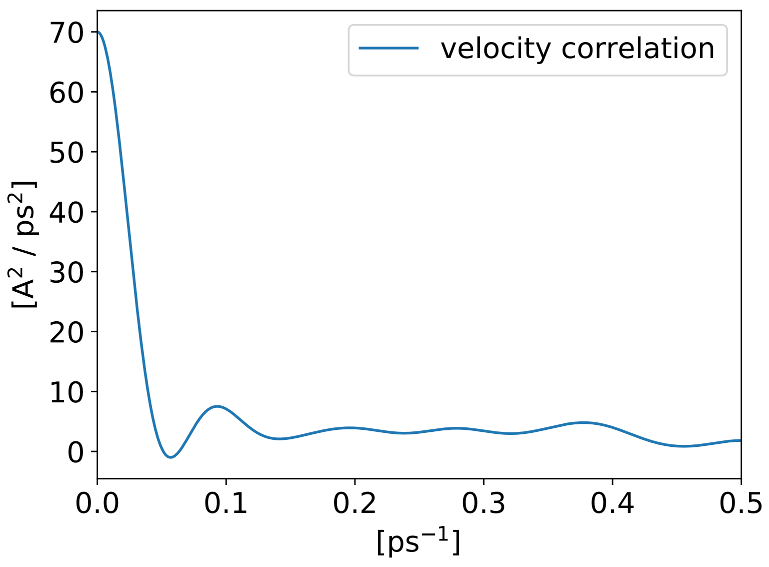

Figure 16: Plot of velocity correlation calculated from (21).

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`

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "initfcc.h"
#include "alpotential.h"
#include "verlet_funcs.h"
#include "utility_funcs.h"
#include "fft_func.h"

/* define some shorthands */
#define kB (8.617e-5) // Boltzmann constant [eV / K]
#define Al_MASS 0.0027964394 // [eV ps^2 / A^2]
#define Nc 4
#define NBR_CELLS (Nc * Nc * Nc)
#define NBR_ATOMS (4 * NBR_CELLS)
#define a0_LIST_LENGTH 45

/* declare some global variables */
double a0;


void task1() {
    double positions[NBR_ATOMS][3]; // [A]
    double a0; // [A]
    double a0_list[a0_LIST_LENGTH];
    double potential_energy; // [eV]

    /*declare variable to sum over */
    int i;

    FILE *potential_energy_file;
    FILE *a0_file;

    /* Assign list of unit cell lengths */
    for (i = 0; i < a0_LIST_LENGTH; i++) {
        a0_list[i] = pow(64.0 + 0.1 * i, 1 / 3.0); // in order to match Fig 1
    }

    potential_energy_file = fopen("potential_energy.txt", "w");
    a0_file = fopen("a0.txt", "w");

    /* Calculate potential energies */
    for (i = 0; i < a0_LIST_LENGTH; i++) {
        a0 = a0_list[i];
        init_fcc(positions, Nc, a0);
        potential_energy = get_energy_AL(positions, Nc * a0, NBR_ATOMS);

        /* write energies to file */
        fprintf(potential_energy_file, "%e\n", potential_energy / NBR_CELLS); //↩
              potential energy per unit cell
        fprintf(a0_file, "%e\n", a0);
    }

    fclose(potential_energy_file);
    fclose(a0_file);
}


void task2() {
    double positions [NBR_ATOMS][3]; // [A]
    double velocities [NBR_ATOMS][3]; // [A / ps]
    double accelerations [NBR_ATOMS][3]; // [A / ps^2]
    double forces [NBR_ATOMS][3]; // [eV / A]
    double disturbance; // [A]
    int i, j; // variables to sum over

    /* Initiate some variables */
    int nbr_timesteps = 1e3;
    double dt = 1e-3; // [ps]
    a0 = pow(66.0, 1 / 3.0); // [A]

    /* Allocate some memory & declare some files for storing result from Verlet ↩
         algorithm */
    double *potential_energies = malloc(nbr_timesteps * sizeof(double));
    double *kinetic_energies = malloc(nbr_timesteps * sizeof(double));
    FILE *potential_energies_file;
    FILE *kinetic_energies_file;
    FILE *parameters_file;


    /* Initiate positions & introduce disturbances */
    init_fcc(positions, Nc, a0);
    for (i = 0; i < NBR_ATOMS; i++){
        for (j = 0; j < 3; j++) {
            disturbance = 2 * 0.065 * (0.5 - (double) rand() / (double) RAND_MAX↩
                 ) * a0; // +-6.5% of a0 in each direction
            positions[i][j] += disturbance;
        }
    }

    /* Initiate velocities & accelerations */
    get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
    init_vel_acc(velocities, accelerations, forces);

    /* Perform velocity Verlet algorithm */
    for (i = 0; i < nbr_timesteps; i++) {
        /* Calculate & store energies */
        potential_energies[i] = get_energy_AL(positions, Nc * a0, NBR_ATOMS);
        kinetic_energies[i] = get_kinetic_energy(velocities, NBR_ATOMS);

        /* Update positions, velocities, & accelerations using forces */
        verlet_single(positions, velocities, accelerations, forces, dt, a0);
    }


    /* write energies to file */
    potential_energies_file = fopen("potential_energies_task2.bin", "w");
    kinetic_energies_file = fopen("kinetic_energies_task2.bin", "w");
    parameters_file = fopen("parameters_task2.bin", "w");

    fwrite(potential_energies, sizeof(double), nbr_timesteps, ↩
         potential_energies_file);
    fwrite(kinetic_energies, sizeof(double), nbr_timesteps, ↩
         kinetic_energies_file);
    double paramList[4] = {nbr_timesteps, dt, NBR_ATOMS, a0};
    int nbr_parameters = 4;
    fwrite(paramList, nbr_parameters * sizeof(double), 1, parameters_file);

    fclose(potential_energies_file);
    fclose(kinetic_energies_file);
    fclose(parameters_file);

    free(potential_energies);
    free(kinetic_energies);
}


void task3() {
    double positions[NBR_ATOMS][3]; // [A]
    double velocities [NBR_ATOMS][3]; // [A / ps]
    double accelerations [NBR_ATOMS][3]; // [A / ps^2]
    double forces [NBR_ATOMS][3]; // [eV / A]
    double disturbance; // [A]
    double temperature = 0.0; // [K]
    double pressure = 0.0; // []
    int i, j; // variables to sum over

    /* Initiate some variables */
    int nbr_timesteps_per_epoch = 1;
    int nbr_epochs = 1e4;
    double dt = 1e-3;
    a0 = pow(66.0, 1 / 3.0);
    double T_eq = 273.15 + 500.0; // [K]
    double P_eq = 6.242e-7; // 1 bar in [eV / A^3]
    double T_decay_constant = 100.0 * dt; // [ps]
    double P_decay_constant = 100.0 * dt; // [ps]

    /* Variables for writing to file */
    double temperatures[nbr_epochs];
    double pressures[nbr_epochs];
    double parameters[6] = {nbr_timesteps_per_epoch, nbr_epochs, ↩
        T_decay_constant, P_decay_constant, a0, dt};
    int nbr_parameters = 6;
    FILE *temperatures_file;
    FILE *pressures_file;
    FILE *parameters_file;
    FILE *final_positions_file;
    FILE *final_velocities_file;


    /* Initiate positions & introduce disturbances */
    init_fcc(positions, Nc, a0);
    for (i = 0; i < NBR_ATOMS; i++) {
        for (j = 0; j < 3; j++) {
            disturbance = 2 * 0.065 * (0.5 - (double) rand() / (double) RAND_MAX↩
                ) * a0; // +-6.5% of a0 in each direction
            positions[i][j] += disturbance;
        }
    }

    /* Initiate velocities & accelerations */
    get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
    init_vel_acc(velocities, accelerations, forces);

    /* Equilibrate the lattice */
    for (i = 1; i < nbr_epochs + 1; i++) {

        /* Perform one epoch of velocity Verlet algorithm.
        temperature & pressure are calculated from time averages. */
        verlet_task3(temperature, pressure, positions, velocities, accelerations↩
            , a0, dt, nbr_timesteps_per_epoch);

        /* Update a0, positions, & velocities */
        temperature = get_temperature(velocities, NBR_ATOMS);
        pressure = get_pressure(a0, positions, velocities, NBR_ATOMS);
        equilibration_update(&a0, positions, velocities, temperature, pressure, ↩
            T_eq, P_eq, T_decay_constant, P_decay_constant, dt);

        /* Save temperatures & pressures for later */
        temperatures[i-1] = temperature;
        pressures[i-1] = pressure;
    }

    /* Write relevant info to file */
    temperatures_file = fopen("temperatures_task3.bin", "w"); // to be used in ↩
        plot
    pressures_file = fopen("pressures_task3.bin", "w"); // to be used in plot
    parameters_file = fopen("parameters_task3.bin", "w"); // to be used in plot
    final_positions_file = fopen("final_positions_task3.bin", "w"); // to be ↩
        used in task 5
    final_velocities_file = fopen("final_velocities_task3.bin", "w"); // to be ↩
        used in task 5

    fwrite(temperatures, sizeof(double), nbr_epochs, temperatures_file);
    fwrite(pressures, sizeof(double), nbr_epochs, pressures_file);
    fwrite(parameters, nbr_parameters * sizeof(double), 1, parameters_file);
    fwrite(positions, NBR_ATOMS * sizeof(double) * 3, 1, final_positions_file);
    fwrite(velocities, NBR_ATOMS * sizeof(double) * 3, 1, final_velocities_file)↩
        ;

    fclose(temperatures_file);
    fclose(pressures_file);
    fclose(parameters_file);
    fclose(final_positions_file);
    fclose(final_velocities_file);
}


void task4() {
    double positions[NBR_ATOMS][3]; // [A]
    double velocities[NBR_ATOMS][3]; // [A / ps]
    double accelerations[NBR_ATOMS][3]; // [A / ps^2]
    double forces[NBR_ATOMS][3]; // [eV / A]
    double disturbance; // [A]
    double temperature = 0.0; // [K]
    double pressure = 0.0; // []
    int i, j; // variables to sum over

    /* Initiate some variables */
    int nbr_timesteps_per_epoch = 1;
    int nbr_epochs = 1e4;
    double dt = 1e-3;
    a0 = pow(66.0, 1 / 3.0);
    double T_eq = 273.15 + 700.0; // [K]
    double T_pre_eq = 2000; // [K] pre-equilibration temperature for melting the↩
        aluminum
    double P_eq = 6.242e-7; // [eV / A^3]
    double T_decay_constant = 100.0 * dt; // [ps]
    double P_decay_constant = 100.0 * dt; // [ps]

    /* Variables for writing to file */
    double temperatures[nbr_epochs];
    double pressures[nbr_epochs];
    double parameters[6] = {nbr_timesteps_per_epoch, nbr_epochs, ↩
        T_decay_constant, P_decay_constant, a0, dt};
    FILE *temperatures_file;
    FILE *pressures_file;
    FILE *parameters_file;
    FILE *final_positions_file;
    FILE *final_velocities_file;


    /* Initiate positions & introduce disturbances */
    init_fcc(positions, Nc, a0);
    for (i = 0; i < NBR_ATOMS; i++) {
        for (j = 0; j < 3; j++) {
            disturbance = 2 * 0.065 * (0.5 - (double) rand() / (double) RAND_MAX↩
                ) * a0; // +-6.5% of a0 in each direction
            positions[i][j] += disturbance;
        }
    }

    /* Initiate velocities & accelerations */
    get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
    init_vel_acc(velocities, accelerations, forces);

    /* Pre-equilibrate the lattice temperature so that we're certian we're in ↩
        liquid phase */
    for (i = 0; i < nbr_epochs / 10; i++) {

        /* Perform one epoch of velocity Verlet algorithm.
        temperature & pressure are calculated from time averages. */
        verlet_task3(temperature, pressure, positions, velocities, accelerations↩
            , a0, dt, nbr_timesteps_per_epoch);

        /* Update a0, positions, & velocities */
        temperature = get_temperature(velocities, NBR_ATOMS);
        pressure = get_pressure(a0, positions, velocities, NBR_ATOMS);
        equilibration_update(&a0, positions, velocities, temperature, pressure, ↩
            T_pre_eq, P_eq, T_decay_constant, P_decay_constant, dt);
    }


    /* Equilibrate the lattice */
    for (i = 0; i < nbr_epochs; i++) {

        /* Perform one epoch of velocity Verlet algorithm.
        temperature & pressure are calculated from time averages. */
        verlet_task3(temperature, pressure, positions, velocities, accelerations↩
            , a0, dt, nbr_timesteps_per_epoch);

        /* Update a0, positions, & velocities */
        temperature = get_temperature(velocities, NBR_ATOMS);
        pressure = get_pressure(a0, positions, velocities, NBR_ATOMS);
        equilibration_update(&a0, positions, velocities, temperature, pressure, ↩
            T_eq, P_eq, T_decay_constant, P_decay_constant, dt);

        /* Save temperatures & pressures for later */
        temperatures[i] = temperature;
        pressures[i] = pressure;
    }

    /* Write relevant info to file */
    temperatures_file = fopen("temperatures_task4.bin", "w"); // to be used in ↩
        plot
    pressures_file = fopen("pressures_task4.bin", "w"); // to be used in plot
    parameters_file = fopen("parameters_task4.bin", "w"); // to be used in plot
    final_positions_file = fopen("final_positions_task4.bin", "w"); // to be ↩
        used in task 5
    final_velocities_file = fopen("final_velocities_task4.bin", "w"); // to be ↩
        used in task 5

    fwrite(temperatures, sizeof(double), nbr_epochs, temperatures_file);
```

```c
    fwrite(pressures, sizeof(double), nbr_epochs, pressures_file);
    fwrite(parameters, sizeof(parameters), 1, parameters_file);
    fwrite(positions, NBR_ATOMS * sizeof(double) * 3, 1, final_positions_file);
    fwrite(velocities, NBR_ATOMS * sizeof(double) * 3, 1, final_velocities_file);

    fclose(temperatures_file);
    fclose(pressures_file);
    fclose(parameters_file);
    fclose(final_positions_file);
    fclose(final_velocities_file);
}


void task5() {
    int nbr_timesteps = 1000;
    double dt = 1e-3; // [ps]
    double parameters[2] = {dt, nbr_timesteps};
    int nbr_parameters = 2;
    int i, j;

    double positions[NBR_ATOMS][3]; // [A]
    double velocities[NBR_ATOMS][3]; // [A / ps]
    double accelerations[NBR_ATOMS][3]; // [A / ps^2]
    double forces[NBR_ATOMS][3]; // [eV / A]
    double initial_parameters[6];

    /* Allocate some memory for storing in each timestep */
    double *mean_squared_displacements = malloc(nbr_timesteps * sizeof(double));
    double ***positions_list = (double ***)malloc(NBR_ATOMS * sizeof(double **)); // positions_list[nbr_atoms][3][2 * nbr_timesteps]
    for (i = 0; i < NBR_ATOMS; i++) {
        positions_list[i] = (double **)malloc(3 * sizeof(double *));
        positions_list[i][0] = (double *)malloc(2 * nbr_timesteps * sizeof(double));
        positions_list[i][1] = (double *)malloc(2 * nbr_timesteps * sizeof(double));
        positions_list[i][2] = (double *)malloc(2 * nbr_timesteps * sizeof(double));
    }

    FILE *initial_positions_file;
    FILE *initial_velocities_file;
    FILE *initial_parameters_file;
    FILE *parameters_file;
    FILE *mean_squared_displacements_file;

    /* Initiate positions & velocities */
    /* The final positions & velocities from Task 3/4 are tuned to have temperature 500/700 C & pressure 1 bar */
    initial_positions_file = fopen("final_positions_task4.bin", "r");
    initial_velocities_file = fopen("final_velocities_task4.bin", "r");
    initial_parameters_file = fopen("parameters_task4.bin", "r");

    int tmp; // this is a bodgy way of suppressing warning about fread:s return value unused
    tmp = fread(positions, NBR_ATOMS * sizeof(double) * 3, 1, initial_positions_file);
    tmp = fread(velocities, NBR_ATOMS * sizeof(double) * 3, 1, initial_velocities_file);
    tmp = fread(initial_parameters, nbr_parameters * sizeof(double), 1, initial_parameters_file);
    tmp++;

    fclose(initial_positions_file);
    fclose(initial_velocities_file);
    fclose(initial_parameters_file);


    /* Initiate forces & accelerations */
    a0 = initial_parameters[4];
    get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
    for (i = 0; i < NBR_ATOMS; i++) {
        accelerations[i][0] = forces[i][0] / Al_MASS;
        accelerations[i][1] = forces[i][1] / Al_MASS;
        accelerations[i][2] = forces[i][2] / Al_MASS;
    }

    /* Perform velocity Verlet algorithm */
    for (i = 1; i < 2 * nbr_timesteps; i++) {
        /* Update positions, velocities, & accelerations using forces */
        verlet_single(positions, velocities, accelerations, forces, dt, a0);

        /* Store positions */
        for (j = 0; j < NBR_ATOMS; j++) {
            positions_list[j][0][i] = positions[j][0];
            positions_list[j][1][i] = positions[j][1];
            positions_list[j][2][i] = positions[j][2];
        }
    }

    /* Calculate mean squared displacements */
    for (i = 0; i < nbr_timesteps; i++) {
        mean_squared_displacements[i] = mean_squared_displacement(i, ↩
             positions_list, dt, nbr_timesteps);
    }

    /* Write mean squared displacement to file */
    mean_squared_displacements_file = fopen("mean_squared_displacements_task5.↩
         bin", "w");
    parameters_file = fopen("parameters_task5.bin", "w");

    fwrite(mean_squared_displacements, nbr_timesteps * sizeof(double), 1, ↩
         mean_squared_displacements_file);
    fwrite(parameters, nbr_parameters * sizeof(double), 1, parameters_file);

    fclose(mean_squared_displacements_file);
    fclose(parameters_file);

    free(mean_squared_displacements);
    free(positions_list);
}


void task6() {
    int nbr_timesteps = 1000;
    double dt = 1e-3; // [ps]
    double parameters[2] = {dt, nbr_timesteps};
    int i, j;

    double positions[NBR_ATOMS][3]; // [A]
    double velocities[NBR_ATOMS][3]; // [A / ps]
    double accelerations[NBR_ATOMS][3]; // [A / ps^2]
    double forces[NBR_ATOMS][3]; // [eV / A]
    double initial_parameters[6];
    int nbr_initial_parameters = 6;
    int nbr_parameters = 6;

    /* Allocate some memory for storing in each timestep */
    double *velocity_correlations = malloc(nbr_timesteps * sizeof(double));
    double ***velocities_list = (double ***)malloc(NBR_ATOMS * sizeof(double **)↩
         ); // velocities_list[nbr_atoms][3][2 * nbr_timesteps]
    for (i = 0; i < NBR_ATOMS; i++) {
        velocities_list[i] = (double **)malloc(3 * sizeof(double *));
        velocities_list[i][0] = (double *)malloc(2 * nbr_timesteps * sizeof(↩
             double));
        velocities_list[i][1] = (double *)malloc(2 * nbr_timesteps * sizeof(↩
             double));
        velocities_list[i][2] = (double *)malloc(2 * nbr_timesteps * sizeof(↩
             double));
    }

    FILE *initial_positions_file;
    FILE *initial_velocities_file;
    FILE *initial_parameters_file;
    FILE *velocities_file;
    FILE *velocity_correlation_file;
    FILE *parameters_file;

    /* Initiate positions & velocities */
    /* The final positions & velocities from Task 3/4 are tuned to have ↩
         temperature 500/700 C & pressure 1 bar */
    initial_positions_file = fopen("final_positions_task4.bin", "r");
    initial_velocities_file = fopen("final_velocities_task4.bin", "r");
    initial_parameters_file = fopen("parameters_task4.bin", "r");

    int tmp; // this is a bodgy way of suppressing warning about fread:s return ↩
         value unused
    tmp = fread(positions, NBR_ATOMS * sizeof(double) * 3, 1, ↩
         initial_positions_file);
    tmp = fread(velocities, NBR_ATOMS * sizeof(double) * 3, 1, ↩
         initial_velocities_file);
    tmp = fread(initial_parameters, nbr_initial_parameters * sizeof(double), 1, ↩
         initial_parameters_file);
    tmp++;

    fclose(initial_positions_file);
    fclose(initial_velocities_file);
    fclose(initial_parameters_file);


    /* Initiate forces & accelerations */
    a0 = initial_parameters[4];
    get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
    for (i = 0; i < NBR_ATOMS; i++) {
        accelerations[i][0] = forces[i][0] / Al_MASS;
        accelerations[i][1] = forces[i][1] / Al_MASS;
        accelerations[i][2] = forces[i][2] / Al_MASS;
    }

    /* Perform velocity Verlet algorithm */
    for (i = 1; i < 2 * nbr_timesteps; i++) {
        /* Update positions, velocities, & accelerations using forces */
        verlet_single(positions, velocities, accelerations, forces, dt, a0);

        /* Store velocities */
        for (j = 0; j < NBR_ATOMS; j++) {
            velocities_list[j][0][i] = velocities[j][0];
            velocities_list[j][1][i] = velocities[j][1];
            velocities_list[j][2][i] = velocities[j][2];
        }
    }

    /* Calculate velocity_correlation */
    for (i = 0; i < nbr_timesteps; i++) {
        velocity_correlations[i] = velocity_correlation(i, velocities_list, dt, nbr_timesteps);
    }

    /* Write velocity correlation to file */
    velocities_file = fopen("velocities_task6.bin", "w");
    velocity_correlation_file = fopen("velocity_correlations_task6.bin", "w");
    parameters_file = fopen("parameters_task6.bin", "w");

    for (i = 0; i < NBR_ATOMS; i++) { // we have to write each list by itself to file
        for (j = 0; j < 3; j++) {
            fwrite(velocities_list[i][j], nbr_timesteps * 2 * sizeof(double), 1, velocities_file);
        }
    }
    fwrite(velocity_correlations, nbr_timesteps * sizeof(double), 1, velocity_correlation_file);
    fwrite(parameters, nbr_parameters * sizeof(double), 1, parameters_file);

    fclose(velocities_file);
    fclose(velocity_correlation_file);
    fclose(parameters_file);

    free(velocity_correlations);
    free(velocities_list);
}


void task7() {
    int nbr_timesteps = 1000; // ought to be the same as in task 6 (BODGE)
    double dt = 1e-3; // [ps] ought to be the same as in task 6 (BODGE)
    // double parameters[2] = {dt, nbr_timesteps};
    int i, j, k; // for summing over

    /* Allocate some memory for storing in each timestep */
    // double *velocity_correlations = malloc(nbr_timesteps * sizeof(double));
    double ***velocities_list = (double ***)malloc(NBR_ATOMS * sizeof(double **)); // velocities_list[nbr_atoms][3][2 * nbr_timesteps]
    for (i = 0; i < NBR_ATOMS; i++) {
        velocities_list[i] = (double **)malloc(3 * sizeof(double *));
        velocities_list[i][0] = (double *)malloc(2 * nbr_timesteps * sizeof(double));
        velocities_list[i][1] = (double *)malloc(2 * nbr_timesteps * sizeof(double));
        velocities_list[i][2] = (double *)malloc(2 * nbr_timesteps * sizeof(double));
    }

    /* arrays for FFT */
    double *tmp_array = malloc(nbr_timesteps * sizeof(double));
    double *powspec = malloc(nbr_timesteps * sizeof(double));
    double *total_powspec = malloc(nbr_timesteps * sizeof(double));
    for(k = 0; k < NBR_ATOMS; k++) { // initiate total_powspec
        total_powspec[k] = 0.0;
    }

    double *freq = malloc(nbr_timesteps * sizeof(double));


    FILE *velocities_file;
    FILE *powerspectrum_file;
    FILE *frequencies_file;

    /* Read velocities from task 6 from file */
    velocities_file = fopen("velocities_task6.bin", "r");
    int tmp;
    for (i = 0; i < NBR_ATOMS; i++) {
        for (j = 0; j < 3; j++) {
            tmp = fread(velocities_list[i][j], nbr_timesteps * 2 * sizeof(double), 1, velocities_file);
        }
    }
    tmp++;
    fclose(velocities_file);

    /* make FFT (powerspectrum) */
    for(i = 0; i < NBR_ATOMS; i++) {
        for(j = 0; j < 3; j++) {
            tmp_array = velocities_list[i][j];
            powerspectrum(tmp_array, powspec, nbr_timesteps); // fft is same as inverse fft here since our function is even
            for(k = 0; k < NBR_ATOMS; k++) {
                total_powspec[k] += powspec[k] / NBR_ATOMS;
            }
        }
    }
    // powerspectrum_shift(total_powspec, nbr_timesteps);
    fft_freq(freq, dt, nbr_timesteps);
    // fft_freq_shift(freq, dt, nbr_timesteps);


    /* save powerspectrum for plotting */
    powerspectrum_file = fopen("powerspectrum_task7.bin", "w");
    frequencies_file = fopen("frequencies_task7.bin", "w");

    fwrite(total_powspec, nbr_timesteps * sizeof(double), 1, powerspectrum_file);
    fwrite(freq, nbr_timesteps * sizeof(double), 1, frequencies_file);
}


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

    // task1();

    // task2();

    // task3();

    // task4();

    // task5();

    // task6();

    task7();

   return 0;
}
```

A.2 Verlet algorithms file verlet_funcs.c

```c
#include "verlet_funcs.h"

#include <math.h>
#include "utility_funcs.h"
#include "alpotential.h"

/* define some shorthands (same as in main) */
#define kB (8.617e-5) // Boltzmann constant [eV / K]
#define Al_MASS 0.0027964394 // [eV ps^2 / A^2]
#define Nc 4
#define NBR_CELLS (Nc * Nc * Nc)
#define NBR_ATOMS (4 * NBR_CELLS)
#define a0_LIST_LENGTH 45


/* Shorthand for initiating velocities & accelerations */
void init_vel_acc(double velocities[][3], double accelerations[][3], double forces[][3]) {
    int i;
    for (i = 0; i < NBR_ATOMS; i++) {
        velocities[i][0] = 0.0;
        velocities[i][1] = 0.0;
        velocities[i][2] = 0.0;
        accelerations[i][0] = forces[i][0] / Al_MASS;
        accelerations[i][1] = forces[i][1] / Al_MASS;
        accelerations[i][2] = forces[i][2] / Al_MASS;
    }
}


/*
    Run Verlet algorithm 1 time.
    Updates positions, velocities, & accelerations using volicities, accelerations, forces, & timestep.
*/
void verlet_single(double positions[][3], double velocities[][3], double accelerations[][3], double forces[][3], double timestep, double a0) {
    int i;
    for (i = 0; i < NBR_ATOMS; i++) {
        /* v(t+dt/2) */
        velocities[i][0] += timestep * 0.5 * accelerations[i][0];
        velocities[i][1] += timestep * 0.5 * accelerations[i][1];
        velocities[i][2] += timestep * 0.5 * accelerations[i][2];

        /* q(t+dt) */
        positions[i][0] += timestep * velocities[i][0];
        positions[i][1] += timestep * velocities[i][1];
        positions[i][2] += timestep * velocities[i][2];
    }

    /* Update forces */
    get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);

    for (i = 0; i < NBR_ATOMS; i++) {
        /* a(t+dt) */
        accelerations[i][0] = forces[i][0] / Al_MASS;
        accelerations[i][1] = forces[i][1] / Al_MASS;
        accelerations[i][2] = forces[i][2] / Al_MASS;

        /* v(t+dt) */
        velocities[i][0] += timestep * 0.5 * accelerations[i][0];
        velocities[i][1] += timestep * 0.5 * accelerations[i][1];
        velocities[i][2] += timestep * 0.5 * accelerations[i][2];
    }
}


/*
    Run Verlet algorithm nbr_timesteps times.
    Updates positions, velocities, & accelerations using forces & timestep.
*/
void verlet_many(double positions[][3], double velocities[][3], double ↩
     accelerations[][3], double a0, double timestep, int nbr_timesteps) {
    /* Initiate forces */
    double forces[NBR_ATOMS][3];
    get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);

    /* Run Verlet algorithm */
    int i;
    for (i = 0; i < nbr_timesteps; i++){
        /* Update positions, velocities, & accelerations using forces */
        verlet_single(positions, velocities, accelerations, forces, timestep, a0↩
             );

        /* Update forces */
        get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);
    }
}


/*
    Run Verlet algorithm nbr_timesteps times.
    Updates positions, velocities, & accelerations using forces & timestep.
    Stores time averages of temperature & pressure in temperature_average & ↩
         pressure_average.
*/
void verlet_task3(double temperature_average, double pressure_average, double ↩
     positions[][3], double velocities[][3], double accelerations[][3], double ↩
     a0, double timestep, int nbr_timesteps) {
    double temperature;
    temperature_average = 0.0;
    double pressure;
    pressure_average = 0.0;

    /* Initiate forces */
    double forces[NBR_ATOMS][3];
    get_forces_AL(forces, positions, Nc * a0, NBR_ATOMS);

    /* Run Verlet algorithm */
    int i;
    for (i = 0; i < nbr_timesteps; i++){
        /* Update positions, velocities, & accelerations using forces */
        verlet_single(positions, velocities, accelerations, forces, timestep, a0↩
             );

        /* Update temperature & pressure time averages */
        temperature = get_temperature(velocities, NBR_ATOMS);
        pressure = get_pressure(a0, positions, velocities, NBR_ATOMS);
        temperature_average += temperature / (double) nbr_timesteps;
        pressure_average += pressure / nbr_timesteps;
    }
}
```

A.3 Utility functions file utility_funcs.c

```c
#include "utility_funcs.h"

#include <stdio.h>
#include <math.h>
#include "verlet_funcs.h"
#include "alpotential.h"

/* define some shorthands (same as in main) */
#define kB (8.617e-5) // Boltzmann constant [eV / K]
#define Al_MASS 0.0027964394 // [eV ps^2 / A^2]
#define Nc 4
#define NBR_CELLS (Nc * Nc * Nc)
#define NBR_ATOMS (4 * NBR_CELLS)


double square(double a) {
    return a * a;
}


/* Returns total kinetic energy of supercell */
double get_kinetic_energy(double velocities[][3], int velocities_length) {
    /* Declare variable to sum over */
    int i;

    /* Initiate sum of kinetic enrgies */
    double sum = 0.0;

    for (i = 0; i < velocities_length; i++){
        sum += 0.5 * pow(velocities[i][0], 2) * Al_MASS;
        sum += 0.5 * pow(velocities[i][1], 2) * Al_MASS;
        sum += 0.5 * pow(velocities[i][2], 2) * Al_MASS;
    }

    return sum;
}


/* Returns temperature of supercell */
double get_temperature(double velocities[][3], int velocities_length) {
    double average_kinetic_energy;
    double temperature;

    average_kinetic_energy = get_kinetic_energy(velocities, velocities_length) /↩
         velocities_length;
    temperature = average_kinetic_energy * 2.0 / (3.0 * kB);

    return temperature;
}


/* Returns pressure of supercell */
double get_pressure(double a0, double positions[][3], double velocities[][3], ↩
     int positions_length) {
    double volume = a0 * a0 * a0 * Nc * Nc * Nc;

    return ((2.0 / 3.0) * get_kinetic_energy(velocities, positions_length) / ↩
        NBR_ATOMS + get_virial_AL(positions, a0 * Nc, NBR_ATOMS)) / volume;
}


/* Updates a0, positions, & velocities using temperature & pressure to better ↩
     match T_eq & P_eq */
void equilibration_update(double *a0, double positions[][3], double velocities↩
     [][3], double temperature, double pressure, double T_eq, double P_eq, ↩
     double T_decay_constant, double P_decay_constant, double timestep) {
    double alpha_T;
    double alpha_P;
    double kappa_T = 2.1; // [A^3 / eV] isothermal compressibility of aluminum @↩
         300 K
    int i; // variable to sum over

    /* Calculate rescaling factors */
    alpha_T = 1.0 + 2.0 * timestep * (T_eq - temperature) / (T_decay_constant * ↩
        temperature);
    alpha_P = 1.0 - kappa_T * timestep * (P_eq - pressure) / P_decay_constant;

    /* Rescale positions & velocitites in order to get closer to P_eq & T_eq */
    for (i = 0; i < NBR_ATOMS; i++) {
        velocities[i][0] *= pow(alpha_T, 1 / 2.0);
        velocities[i][1] *= pow(alpha_T, 1 / 2.0);
        velocities[i][2] *= pow(alpha_T, 1 / 2.0);

        positions[i][0] *= pow(alpha_P, 1 / 3.0);
        positions[i][1] *= pow(alpha_P, 1 / 3.0);
        positions[i][2] *= pow(alpha_P, 1 / 3.0);
    }
    *a0 *= pow(alpha_P, 1 / 3.0);
}

```

```c

/* Returns mean squared displacement at time t of particles in lattice given an array
    positions_list[nbr_atoms][nbr_dimensions][2 * nbr_timesteps] */
double mean_squared_displacement(int t_index, double ***positions_list, double timestep, int nbr_timesteps) {
    int i, j; // variables to sum over
    double squared_displacements = 0.0; // tmp variable to hold sum

    /* sum up squared displacements over time and particles */
    for (i = 0; i < nbr_timesteps; i++) {
        for (j = 0; j < NBR_ATOMS; j++) {
            squared_displacements += square(positions_list[j][0][i + t_index] - positions_list[j][0][i]);
            squared_displacements += square(positions_list[j][1][i + t_index] - positions_list[j][1][i]);
            squared_displacements += square(positions_list[j][2][i + t_index] - positions_list[j][2][i]);
        }
    }

    return squared_displacements / (double) (NBR_ATOMS * nbr_timesteps); // average by dividing by nbr_atoms & nbr_timesteps
}


double velocity_correlation(int t_index, double ***velocities_list, double timestep, int nbr_timesteps) {
    int i, j; // variables to sum over
    double vel_corr = 0.0; // tmp variable to hold sum

    /* sum up squared displacements over time and particles */
    for (i = 0; i < nbr_timesteps; i++) {
        for (j = 0; j < NBR_ATOMS; j++) {
            vel_corr += velocities_list[j][0][i + t_index] * velocities_list[j][0][i];
            vel_corr += velocities_list[j][1][i + t_index] * velocities_list[j][1][i];
            vel_corr += velocities_list[j][2][i + t_index] * velocities_list[j][2][i];
        }
    }

    return vel_corr / (double) (NBR_ATOMS * nbr_timesteps); // average by dividing by nbr_atoms & nbr_timesteps
}
```

A.4 Aluminum properties file `alpotential.c`

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


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

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


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

#define EMBEDDING_ENERGY_ROWS 13
const double embedding_energy[65] = {0, 0.1000, 0.2000, 0.3000, 0.4000, 0.5000, ↩
     0.6000, 0.7000, 0.8000, 0.9000, 1.0000, 1.1000, 1.2000, 0, -1.1199, ↩
     -1.4075, -1.7100, -1.9871, -2.2318, -2.4038, -2.5538, -2.6224, -2.6570, ↩
     -2.6696, -2.6589, -2.6358, -18.4387, -5.3706, -2.3045, -3.1161, -2.6175, ↩
     -2.0666, -1.6167, -1.1280, -0.4304, -0.2464, -0.0001, 0.1898, 0.2557, ↩
     86.5178, 44.1632, -13.5018, 5.3853, -0.3996, 5.9090, -1.4103, 6.2976, ↩
     0.6785, 1.1611, 1.3022, 0.5971, 0.0612, -141.1819, -192.2166, 62.9570, ↩
     -19.2831, 21.0288, -24.3978, 25.6930, -18.7304, 1.6087, 0.4704, -2.3503, ↩
     -1.7862, -1.7862};


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

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

    int k_lo = 0, k_hi = m;

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

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

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

    return result;
}

/* Evaluates the derivative of the spline in x. */
double splineEvalDiff(double x, const double *table, int m) {
    /*int m = mxGetM(spline), i, k;
   double *table = mxGetPr(spline);
   */
  int i, k;
  double result;

    int k_lo = 0, k_hi = m;

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

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

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

    return result;
}

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

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

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

  rcut = 6.06;
  rcut_sq = rcut * rcut;

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

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

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

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

    densityi = density[i];

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

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

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

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

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

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

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

    densityi = density[i];

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

```c

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

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

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

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

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

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

}

/* Returns the potential energy */
double get_energy_AL(double positions[][3], double cell_length, int nbr_atoms)
{
  int i, j;
  double cell_length_inv, cell_length_sq;
  double rcut, rcut_sq;
  double energy;
  double densityi, dens;
  double sxi, syi, szi, sxij, syij, szij, rij,  rij_sq;

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

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

  rcut = 6.06;
  rcut_sq = rcut * rcut;

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

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

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

  energy = 0;

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

    densityi = density[i];

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

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

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

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

        /* Add energy contribution from i-j interaction */
        energy += splineEval(rij, pair_potential, PAIR_POTENTIAL_ROWS);

      }
    }
    density[i] = densityi;
  }

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

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

  return(energy);

}

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

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

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

  rcut = 6.06;
  rcut_sq = rcut * rcut;

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

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

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

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

    densityi = density[i];

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

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

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

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

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

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

  virial = 0;

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

    densityi = density[i];

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

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

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

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

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

        virial += force * rij_sq;
            }
    }
  }

  virial /= 3.0;

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

  return(virial);

}
```

A.5 FCC initialization file `initfcc.c`

```c
/*
initfcc.c
Program that arranges atoms on a fcc lattice.
Created by Anders Lindman on 2013-03-15.
*/

#include <stdio.h>

/* Function takes a matrix of size [4*N*N*N][3] as input and stores a fcc lattice in it. N is the number of unit cells in each dimension and lattice_param is the lattice parameter. */
void init_fcc(double positions[][3], int N, double lattice_param)
{
    int i, j, k;
    int xor_value;

    for (i = 0; i < 2 * N; i++){
        for (j = 0; j < 2 * N; j++){
            for (k = 0; k < N; k++){
                if (j % 2 == i % 2 ){
                    xor_value = 0;
                }
                else {
                    xor_value = 1;
                }
                positions[i * N * 2 * N + j * N + k][0] = lattice_param * (0.5 * xor_value + k);
                positions[i * N * 2 * N + j * N + k][1] = lattice_param * (j * 0.5);
                positions[i * N * 2 * N + j * N + k][2] = lattice_param * (i * 0.5);
            }
        }
    }
}
```

A.6 plot_task1.py

```python
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
import struct

### for plotting nice plots ###
width = 6
height = width/1.5
fsize = 16
font = {'size': fsize}
mpl.rc('font', **font)
mpl.rc('xtick', labelsize=fsize)
mpl.rc('ytick', labelsize=fsize)
mpl.rc('text', usetex=False)

### read files generated by C ###
potential_energy = np.fromfile('potential_energy.txt', sep='\n')
a0 = np.fromfile('a0.txt', sep='\n')

### plot E(a) ###
fig, ax = plt.subplots()
ax.plot(a0**3, potential_energy, '.')
ax.set_xlabel(r'$a^3$ [A$^3$]')
ax.set_ylabel(r'$E_\mathrm{p}$ [eV / unit cell]')

#plt.legend()
plt.tight_layout()
filename = f'plot_task1.pdf'
plt.savefig(filename)
plt.show()
```

A.7 plot_task2.py

```python
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
import struct

### for plotting nice plots ###
width = 6
height = width / 1.5
fsize = 16
font = {'size': fsize}
```

```python
mpl.rc('font', **font)
mpl.rc('xtick', labelsize=fsize)
mpl.rc('ytick', labelsize=fsize)
mpl.rc('text', usetex=False)

### read binary files generated by C ###
potential_energies = np.fromfile('potential_energies_task2.bin')
kinetic_energies = np.fromfile('kinetic_energies_task2.bin')
total_energies = potential_energies + kinetic_energies
parameters = np.fromfile('parameters_task2.bin')

### extract parameters used ###
nbr_timesteps = int(parameters[0])
dt = parameters[1]
nbr_atoms = parameters[2]
nbr_cells = nbr_atoms / 4
a0 = parameters[3]
t = np.array([i * dt for i in range(0, nbr_timesteps)])

### prepare broken axis ###
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111) # The big subplot
ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212)

# Turn off axis lines and ticks of the big subplot
ax.spines['top'].set_color('none')
ax.spines['bottom'].set_color('none')
ax.spines['left'].set_color('none')
ax.spines['right'].set_color('none')
ax.set_xticks([])
ax.set_yticks([])

# do some other stuff
ax1.spines['bottom'].set_visible(False)
ax2.spines['top'].set_visible(False)
ax1.set_xticks([])
ax2.xaxis.tick_bottom()
d = .015  # how big to make the diagonal lines in axes coordinates

# arguments to pass to plot, just so we don't keep repeating them
kwargs = dict(transform=ax1.transAxes, color='k', clip_on=False)
ax1.plot((-d, +d), (-d, +d), **kwargs)        # top-left diagonal
ax1.plot((1 - d, 1 + d), (-d, +d), **kwargs)  # top-right diagonal
kwargs.update(transform=ax2.transAxes)  # switch to the bottom axes
ax2.plot((-d, +d), (1 - d, 1 + d), **kwargs)  # bottom-left diagonal
ax2.plot((1 - d, 1 + d), (1 - d, 1 + d), **kwargs)  # bottom-right diagonal

### plot E(t) ###
l0 = ax2.plot(t, potential_energies / nbr_cells, label=r'potential energy', ↩
     color='C0')
l1 = ax1.plot(t, kinetic_energies / nbr_cells, label=r'kinetic energy', color='↩
     C1')
l2 = ax2.plot(t, total_energies / nbr_cells, label=r'total energy', color='C2')
ax.set_xlabel('\n' + r'$t$ [ps]')
ax.set_ylabel(r'$E$ [eV / unit cell]' + '\n\n\n')
ax1.set_ylim(top=0.7, bottom=-0.3)
ax1.set_yticks([0.6, 0.4, 0.2, 0.0])
ax2.set_yticks([-12.8, -13.0, -13.2])
ax2.set_ylim(top=-12.3, bottom=-13.3)


### plot 800 K line ###
kB = 8.617e-5 # Boltzmann constant [eV / K]
E_800K = (3 / 2) * kB * 800 * 4 # *4 since 4 atoms per unit cell
l3 = ax1.plot([min(t), max(t)], [E_800K, E_800K], 'r--', zorder=-1, label=r'↩
     kinetic energy at 800 K')

### legend & save file ###
fig.legend((l0[0], l1[0], l2[0], l3[0]),
            (r'potential energy', r'kinetic energy', r'total energy', r'kinetic ↩
                 energy at 800 K'),
            loc=(0.38, 0.48))
plt.tight_layout()
filename = f'plot_task2_dt=' + str(dt) + '.pdf'
plt.savefig(filename, bbox_inches='tight')
plt.show()
```

A.8 plot_task3_and_4 .py

```python
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
import struct

### for plotting nice plots ###
```

```
width = 6
height = width / 1.5
fsize = 16
font = {'size': fsize}
mpl.rc('font', **font)
mpl.rc('xtick', labelsize=fsize)
mpl.rc('ytick', labelsize=fsize)
mpl.rc('text', usetex=False)

### read binary files generated by C ###
temperatures_solid = np.fromfile('temperatures_task3.bin')
pressures_solid = np.fromfile('pressures_task3.bin')
parameters_solid = np.fromfile('parameters_task3.bin')
temperatures_liquid = np.fromfile('temperatures_task4.bin')
pressures_liquid = np.fromfile('pressures_task4.bin')
parameters_liquid = np.fromfile('parameters_task4.bin')
indices_solid = np.array([i for i in range(len(temperatures_solid))])
indices_liquid = np.array([i for i in range(len(temperatures_liquid))])

### extract parameters ###
nbr_timesteps_per_epoch = int(parameters_solid[0])
nbr_epochs = int(parameters_solid[1])
T_decay_constant = parameters_solid[2]
P_decay_constant = parameters_solid[3]
timestep = parameters_solid[5]

### plot T(t) & P(t) ###
fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(8, 8))
ax1.plot(timestep * indices_solid, temperatures_solid, label=r'solid phase', ↩
     color='C0')
ax1.plot(timestep * indices_liquid, temperatures_liquid, label=r'liquid phase', ↩
     color='C1')
ax1.set_xlabel(r'$t$ [ps]')
ax1.set_ylabel(r'$T$ [K]')
ax2.plot(timestep * indices_solid, pressures_solid, label=r'solid phase', color=↩
     'C0')
ax2.plot(timestep * indices_liquid, pressures_liquid, label=r'liquid phase', ↩
     color='C1')
ax2.set_xlabel(r'$t$ [ps]')
ax2.set_ylabel(r'$P$ [eV / A$^3$]')

### plot 500 C line ###
T_eq_solid = 500 + 273.15
P_eq_solid = 6.242e-7
T_eq_liquid = 700 + 273.15
P_eq_liquid = 6.242e-7
ax1.plot(timestep * indices_solid, T_eq_solid * np.ones(indices_solid.shape), '↩
     --', zorder=-1, label=r'500 $^\mathrm{o}$C', color='k')
ax1.plot(timestep * indices_liquid, T_eq_liquid * np.ones(indices_liquid.shape),↩
      ':', zorder=-1, label=r'700 $^\mathrm{o}$C', color='k')
ax2.plot(timestep * indices_solid, P_eq_solid * np.ones(indices_solid.shape), '↩
     --', zorder=-1, label=r'1 bar', color='k')

### legend & save file ###
ax1.legend(loc=1)
ax2.legend()
plt.tight_layout()
filename = f'plot_task3_and_4.pdf'
plt.savefig(filename)
plt.show()
```

A.9 plot_task5.py

```
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
import struct

### for plotting nice plots ###
width = 6
height = width / 1.5
fsize = 16
font = {'size': fsize}
mpl.rc('font', **font)
mpl.rc('xtick', labelsize=fsize)
mpl.rc('ytick', labelsize=fsize)
mpl.rc('text', usetex=False)

### read binary files generated by C ###
mean_squared_displacements = np.fromfile('mean_squared_displacements_task5.bin')
parameters = np.fromfile('parameters_task5.bin')

### extract parameters ###
dt = parameters[0]
nbr_timesteps = int(parameters[1])
t = np.array([i * dt for i in range(nbr_timesteps)])
```

```

### plot mean_squared_displacements(t) ###
fig, ax = plt.subplots()
ax.plot(t[1:-1], mean_squared_displacements[1:-1], label=r'mean squared ↩
    displacement')
ax.set_xlabel(r'$t$ [ps]')
ax.set_ylabel(r'[A]')

### legend & save file ###
plt.legend()
plt.tight_layout()
filename = f'plot_task5_timesteps={nbr_timesteps}_dt={dt}.pdf'
plt.savefig(filename)
plt.show()
```

A.10 plot_task6.py

```
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
import struct

### for plotting nice plots ###
width = 6
height = width / 1.5
fsize = 16
font = {'size': fsize}
mpl.rc('font', **font)
mpl.rc('xtick', labelsize=fsize)
mpl.rc('ytick', labelsize=fsize)
mpl.rc('text', usetex=False)

### read binary files generated by C ###
velocity_correlations = np.fromfile('velocity_correlations_task6.bin')
parameters = np.fromfile('parameters_task6.bin')

### extract parameters ###
dt = parameters[0]
nbr_timesteps = int(parameters[1])
t = np.array([i * dt for i in range(nbr_timesteps)])

### plot velocity_correlations(t) ###
fig, ax = plt.subplots()
ax.plot(t, velocity_correlations, label=r'velocity correlation')
ax.set_xlabel(r'$t$ [ps]')
ax.set_ylabel(r'[A$^2$ / ps$^2$]')

### legend & save file ###
plt.legend()
plt.tight_layout()
filename = f'plot_task6.pdf'
plt.savefig(filename)
plt.show()

### calculate cosine transform ###
D = np.trapz(velocity_correlations, x=t) / 3
print(f'D = {D} []')
```

A.11 plot_task6.py

```
import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
import struct

### for plotting nice plots ###
width = 6
height = width / 1.5
fsize = 16
font = {'size': fsize}
mpl.rc('font', **font)
mpl.rc('xtick', labelsize=fsize)
mpl.rc('ytick', labelsize=fsize)
mpl.rc('text', usetex=False)

### read binary files generated by C ###
powerspectrum = np.fromfile('powerspectrum_task7.bin')
frequencies = np.fromfile('frequencies_task7.bin')
correlation = 1j * np.zeros(powerspectrum.shape)
N = len(powerspectrum)
for l in range(N):
    for n in range(N - 1):
        correlation[l] += powerspectrum[n]**2 * np.exp(2 * np.pi * 1j * l * n / ↩
             N) / N

### plot velocity_correlations(t) ###
fig, ax = plt.subplots()
ax.plot(1e-3 * np.array([i for i in range(N)]), np.real(correlation), label=r'↩
     velocity correlation')
ax.set_xlabel(r'[ps$^{-1}$]')
ax.set_ylabel(r'[A$^2$ / ps$^2$]')
ax.set_xlim(left=0.0, right=1e-3 * N / 2)
ax.set_yticks([])

### legend & save file ###
plt.legend()
plt.tight_layout()
filename = f'plot_task7.pdf'
plt.savefig(filename)
plt.show()
```