Computational Physics 2022 - Home assignment 1a

MD Simulation: Static Properties of aluminium in solid and liquid state

Nico Guth & Nastaran Fallah Randjbar

December 9, 2022

Task №	Points	Avail. points
Σ		

0 Introduction

This project focuses on using Molecular Dynamics (MD) simulation techniques and an inter-atomic potential to study static properties of aluminium, both in a solid and a liquid state. Aluminium is a metal with atomic number 13, and at room temperature, it has the face-centered-cubic (fcc) crystal structure. FCC is a crystal cubic structure with atoms arranged at all corners, and centers of each cube face. Overall, there are 4 atoms in one unit cell. One of the most important parameters in crystalline structures is the "lattice constant" a_0. In case of fcc, the lattice constant is the dimension of each side of the unit cell. This is a concept that will be used frequently in this report.

1 Task 1 1/1

1.1 Objective and Procedure

The objective here is to find the lattice constant of Al in equilibrium ($T = 0\,\mathrm{K}$), a task that is done through the calculation of the potential energy with atoms arranged in an ideal fcc structure.

In this report, an Al-supercell of $4 \times 4 \times 4$ unit cells is simulated. An array of dimensions 256×3 is created that holds the position coordinates of all the 256 atoms. The positions are initialized to the ideal arrangement of a fcc crystal of a specific a_0. The total potential energy of this system is then calculated using a metallic bonding pair interaction model.

Based on Figure 1 in the description of the problem, it is expected to find the minimum of the potential energy around $a_0 = \sqrt[3]{65.5\,\text{Å}^3} = 4.03\,\text{Å}$. Therefore, the described steps are repeated for 1000 values of $a_0 \in [3.98, 4.08]$. A quadratic fit is carried out using

$$E_{pot} = c_1 a_0^2 + c_2 a_0 + c_3\,. \tag{1}$$

1.2 Results and discussion

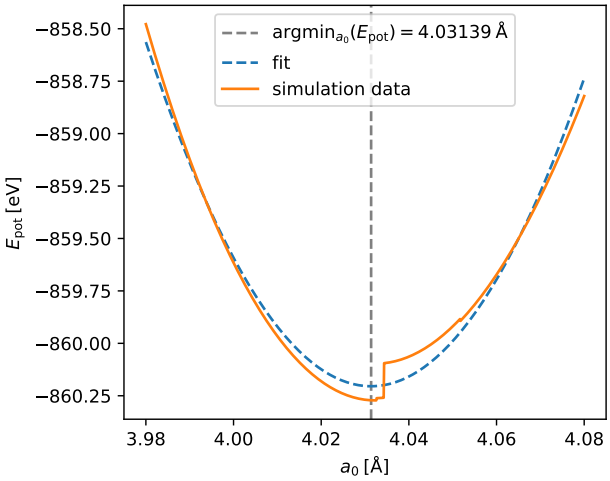

Figure 1: Dependence of the potential energy of an ideal aluminium crystal on the lattice constant. Shown is the calculated potential energy and a quadratic fit of the potential. The minimum of the fit is indicated.

The achieved potential energies at various lattice parameters and the corresponding fit are shown in figure 1. A minimum of the potential energy is achieved for

$$\frac{dE_{pot}}{da_0} = 0 \Rightarrow a_0 = -\frac{1}{2}\frac{c_2}{c_1} = 4.031\,39\,\text{Å} \tag{2}$$

which is the estimated lattice parameter for aluminium at $T = 0\,\mathrm{K}$.

The theoretically calculated value for the atomic van der Waals radius of aluminium is $r = 1.84\,\text{Å}$ [1]. This leads to a theoretical lattice constant of aluminium

$$a_0 = 2 \times r \times \sqrt{2} = 5.204\,\text{Å}\,. \tag{3}$$

An experimentally found value of the lattice constant of aluminium (extrapolated to 0 K) is

$$a_0 = 4.0317\,\text{Å}\,[2] \tag{4}$$

which is very close to the value found in this study. Therefore, the van der Waals radius of aluminium does not seem to be a good value to compare our results to. The shape of the calculated potential suggests a very inappropriate approximation, however, it is sufficient for the purposes of this study.

2 Task 2 3/3

2.1 Objective and Procedure

Here, the objective is to find the most suitable time step size δt of our simulation in order to achieve energy conservation and still be able to cover a long enough timespan. The time evolution of a crystal slightly varied from the ideal structure is calculated for various δt.

According to the lecture notes (section 4.3) [5], "a typical time step for an atomic system is a few femtoseconds". Therefore, in this solution, a time step of $\delta t = 5\,\text{fs}$ is chosen as a starting point. The simulation time is decided to be $t_{\max} = 15\,\text{ps}$ and therefore, for different time step sizes δt the number of time steps $n_{\text{steps}} = t_{\max}/\delta t$ varies.

A crystal is initialized in the same way as in task 1 and a deviation from equilibrium position is introduced by adding a random number ϵ which is uniformly distributed in the range $[-a_0 \cdot 0.065\ ,\ +a_0 \cdot 0.065]$. The initial position coordinates then will be

$$x(t=0) = x(\text{ideal}) + \epsilon \tag{5}$$
$$y(t=0) = y(\text{ideal}) + \epsilon \tag{6}$$
$$z(t=0) = z(\text{ideal}) + \epsilon\,. \tag{7}$$

The initial velocities of all the particles are zero

$$v_x(t=0) = v_y(t=0) = v_z(t=0) = 0\,. \tag{8}$$

These initial conditions are used to study the time evolution of each particle. Using periodic boundary conditions, the potential energy and the forces are obtained. From this force, an acceleration is derived, which is then used in the Velocity Verlet Algorithm (explained in the next section) in order to solve the equation of motion.

Next, the evolution of the kinetic, potential and total energies is plotted. The kinetic energy is calculated as

$$E_{kin}(t) = \sum_{i=1}^{N} \frac{p_i(t)^2}{2m}\,, \tag{9}$$

the potential energy calculated for the new positions and the total energy is the sum $E_{tot} = E_{kin} + E_{pot}$. $m = 26.9815\,\text{u}$ is the mass of an aluminium atom and $p_i = mv_i$ is the momentum of atom i. Additionally, the temperature of the system is calculated as the time average

$$T = \frac{2}{3Nk_B}\left\langle \sum_{i=1}^{N} \frac{p_i(t)^2}{2m} \right\rangle . \tag{10}$$

Here, $N = 256$ is the number of atoms in the system and k_B is the Boltzmann constant.

All the above-mentioned steps are repeated for different time steps and evaluated in terms of energy conservation.

2.2 Velocity Verlet

The Velocity Verlet algorithm is a method of solving the equations of motion for any dynamic physical system. What needs to be done is finding out how $r(t)$ and $r(t+\delta t)$ for one particle are related, provided that all the information about other particles is given. Supposing that δt is sufficiently small and by using Taylor expansion, the relation

$$r(t+\delta t) = 2r(t) + 2r(t-\delta t) + \frac{d^2 r}{dt^2}\delta t^2 + O(\delta t^4) \tag{11}$$

can be derived. This is an algorithm that updates the position, and is called Verlet position integrator. The problem with this integrator is, however, that it is not "self-starting", which means that if the simulation needs to start at t=0, it would need to know the position at $t = 0 - \delta t$.

In order to address this issue and by following a slightly different approach, the following expressions for position and velocity will be obtained:

$$r(t+\delta t) = r(t) + v(t)\delta t + \frac{F(t)}{2m}\delta t \tag{12}$$

$$v(t+\delta t) = v(t) + \frac{F(t+\delta t) + F(t)}{2m}\delta t \tag{13}$$

It is important to notice that $r(t+\delta t)$ must be updated before $v(t+\delta t)$ is updated, which is only rational. In order to apply velocity Verlet in MD with a time step, there are certain steps to be followed:

$$1) \quad v(t+\delta t/2) = v(t) + \frac{1}{2}a(t)\delta t \tag{14}$$

$$2) \quad r(t+\delta t) = r(t) + v(t+\delta t/2)\delta t \tag{15}$$

$$3) \quad \text{Calculate the acceleration } a(t+\delta t) = \frac{F(t+\delta t)}{m} \tag{16}$$

$$4) \quad v(t+\delta t) = v(t+\delta t/2) + \frac{1}{2}a(t+\delta t)\delta t \tag{17}$$

Velocity Verlet is a very convenient and numerically stable algorithm for MD simulations. [4] [5] [7]

2.3 Results and discussion

Shown in Figure 2, 3 and 4 is the time dependence of the energies of simulations at δt that are small enough, a little too large and far too large respectively. Regarding the conservation of total energy, no drift in the total energy is visible for $\delta t = 5\,\text{fs}$ and the fluctuations are reasonably small, therefore the total energy can be considered as conserved. There is a small divergence in the total energy when δt is just a little too large (15 fs), however depending on how long the simulation runs, this divergence can grow much larger. For very large values of δt (50 fs), the total energy diverges very fast.

Regarding the average temperature, the following temperatures are calculated (time average):

$$T(\delta t = 5\,\text{fs}) = 759.35\,\text{K} \qquad T(\delta t = 15\,\text{fs}) = 655.37\,\text{K} \qquad T(\delta t = 50\,\text{fs}) = 4.2064 \times 10^9\,\text{K}$$

Therefore, in the cases where the time step is small enough or just a little too large, the temperature remains in the expected range $600K < T < 850K$, but for the case with a very large time step the temperature takes an unreasonably large value, since the kinetic energy is diverging.

Overall, a good result is obtained using a time step size

$$\delta t = 5\,\text{fs}$$

and this time step is used for all following simulations unless stated otherwise.

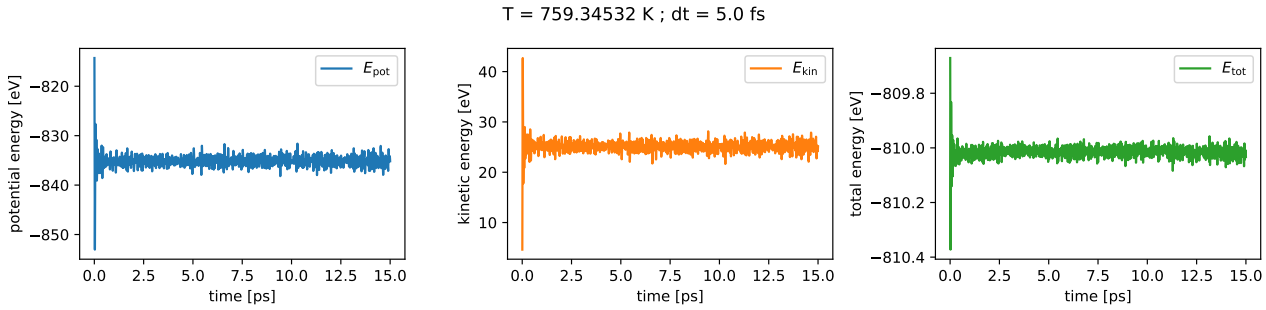

Figure 2: Energy-Time diagram for Aluminium for a $\delta t = 5\,\text{fs}$ that is small enough. Shown is the potential, kinetic and total energy of the system at each simulation timestep.

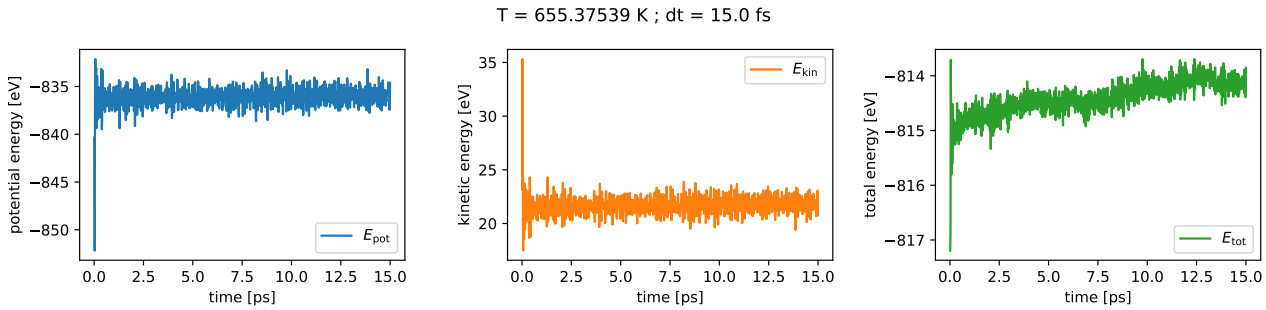

Figure 3: Energy-Time diagram for Aluminium for a $\delta t = 15\,\text{fs}$ that is just a little too large. Shown is the potential, kinetic and total energy of the system at each simulation timestep.

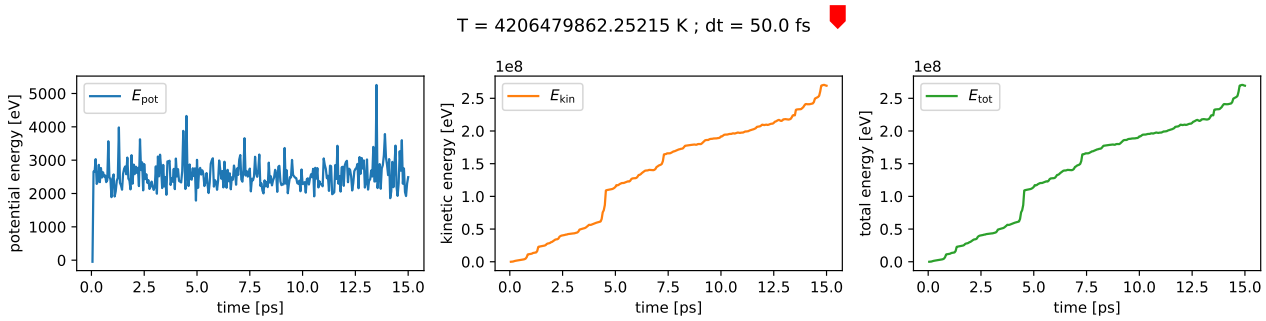

Figure 4: Energy-Time diagram for Aluminium for a $\delta t = 50\,\text{fs}$ that is way too large. Shown is the potential, kinetic and total energy of the system at each simulation timestep.

3 Task 3 4/4

3.1 Objective and Procedure

In this task, initial conditions are sought that correspond to the system being at a particular temperature and pressure. In order to obtain these conditions, some equilibration routines must be performed.

Initial positions and velocities are required for a simulation. These initial conditions correspond to some particular temperature T and pressure P of the system (time average). In order to obtain these macroscopic properties after some equlibration time, a scaling of position and velocity must be performed.

Introducing the instantaneous temperature

$$\mathcal{T}(t) = \frac{2}{3Nk_B} E_{kin}\,, \tag{18}$$

the initial temperature is $\mathcal{T}(0)$. An exponential decay of $\mathcal{T}(t)$ towards a desired temperature T_{eq}

$$\mathcal{T}(t) = T_{eq} + (\mathcal{T}(0) - T_{eq}) \exp\left(\frac{-t}{\tau_T}\right) \tag{19}$$

is achieved when the velocities after each time step in the Velocity Verlet simulation are scaled as

$$v_i^{new} = \alpha_T^{1/2} v_i^{old} \tag{20}$$

with the scaling constant calculated as

$$\alpha_T(t) = 1 + \frac{2\Delta t}{\tau_T} \frac{T_{eq} - \mathcal{T}(t)}{\mathcal{T}(t)}\,. \tag{21}$$

Similarly, the instantaneous pressure

$$\mathcal{P}(t) = \frac{1}{V}\left(Nk_B\mathcal{T}(t) + \mathcal{W}\right) \tag{22}$$

can also be changed so that it decays exponentially to the value of the pressure at equilibrium P_{eq}.

$$\mathcal{P}(t) = P_{eq} + (\mathcal{P}(0) - P_{eq}) \exp\left(\frac{-t}{\tau_P}\right) \tag{23}$$

$\mathcal{W}$ is the virial function. The difference to the temperature decay is that the positions (including the lattice constant) need to be scaled (instead of the velocities) according to

$$r_i^{new} = \alpha_P^{1/3} r_i^{old} \tag{24}$$

with

$$\alpha_P(t) = 1 - \kappa_T \frac{\Delta t}{\tau_P} [P_{eq} - \mathcal{P}(t)] \,. \tag{25}$$

κ_T is the isothermal compressibility, which is taken to be $\kappa_T = 0.013\,85\,\mathrm{GPa}^{-1}$ for aluminium (with the equivalent value of $1.385 \times 10^{-11}\,\mathrm{m}^2/\mathrm{N}$) [6].

Now, in order to do the task, scaling is performed for just the temperature at first. After T_{eq} is obtained, without stopping this procedure (to keep the temperature constant), the scaling for pressure is started. According to the lecture, this split procedure is recommended and makes it easier to obtain equilibrium. When the desired values are achieved, i.e. $T = 500\,^{\circ}\mathrm{C} = 773.15\,\mathrm{K}$ and $P = 1\,\mathrm{bar}$, the obtained initial conditions (lattice constant, position and velocity for each particle) are used to start a simulation (without scaling). This way, it can be verified that a system of the desired temperature and pressure is achieved and is in a state of thermal equilibrium.

3.2 Results and discussion

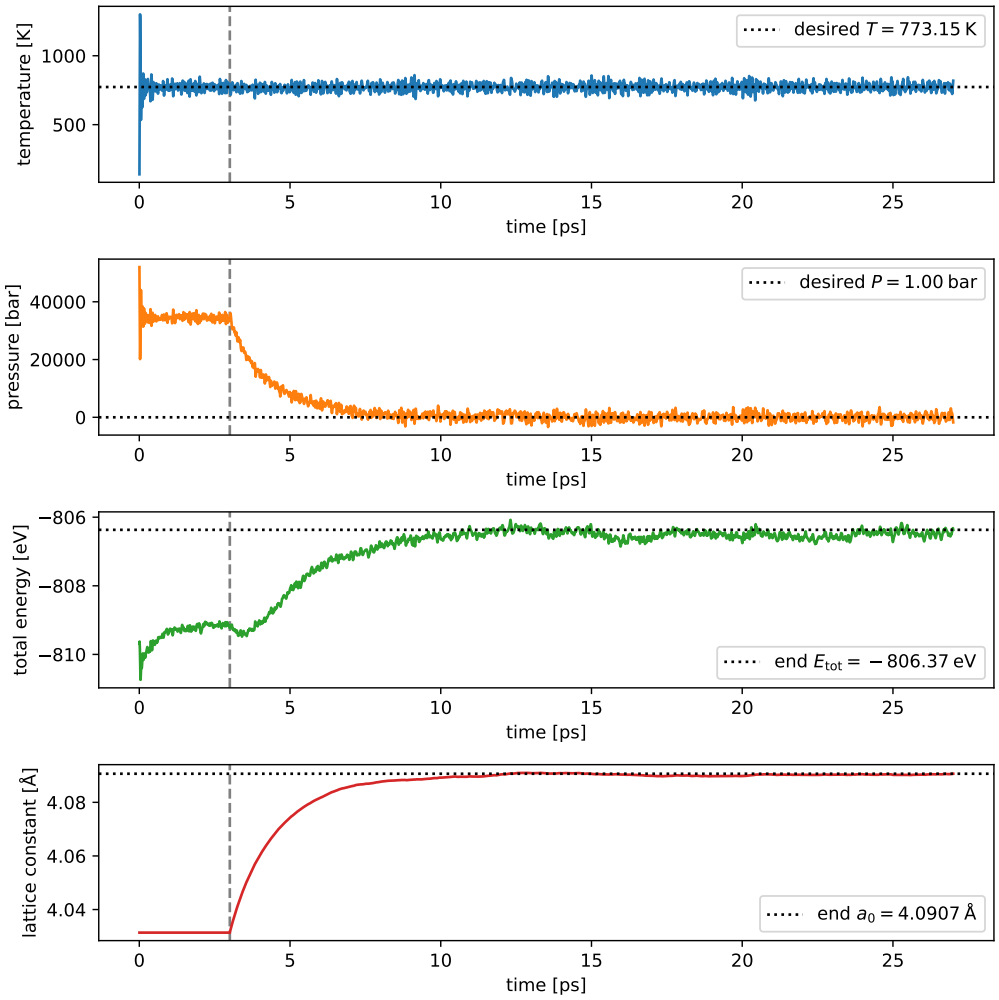

Figure 5: Time evolution of the instantaneous temperature, instantaneous pressure, total energy and lattice constant during the equilibration routine.

In figure 5, the performed equilibration of both the temperature and the pressure are illustrated. At each time step the instantaneous temperature, the instantaneous pressure, the total energy and the lattice constant are shown. The temperature equilibration was done using

$$\delta t = 5\,\mathrm{fs} \quad \tau_T = 100\delta t \quad \Delta t = 3.01\,\mathrm{ps} \tag{26}$$

where δt is the time step size and Δt is the run time of the equilibration. Then, an additional pressure equilibration was performed using

$$\delta t = 5\,\text{fs} \quad \tau_P = 3\tau_T \quad \Delta t = 24.01\,\text{ps}\,. \tag{27}$$

During this equilibration procedure, the temperature and pressure converges towards the desired values. However, due to large fluctuations during the simulation, the exact desired values cannot be reached with a high precision. Especially, the pressure has a high amount of noise and is oscillating around 0 bar.

The total energy and lattice constant (equivalent to volume) of the system also converge during equilibration, which can be seen as a sanity check, that the simulation is physically sensible, because when decreasing pressure at constant temperature, the system has to gain energy and volume.

To further check if the procedure yields an equilibrium state at the desired temperature and pressure, the positions, velocities and the lattice parameter at the end of the equilibration procedure were used to simulate the system. This is shown in fig. 6 and here, of course, the equilibration routines are turned off.

It is shown, that even without the position and velocity scaling, the temperature, pressure and total energy remain essentially constant with some amount of fluctuations. This strongly suggests that the system is in an equilibrium state. The achieved temperature (time average)

$$T = \langle \mathcal{T}(t) \rangle_{\text{time}} = (778.17 \pm 28.03)\,\text{K}$$

is reasonably close to the desired temperature (773.15 K). However, the pressure fluctuations overshadow the achieved pressure

$$P = \langle \mathcal{P}(t) \rangle_{\text{time}} = (-172.51 \pm 1220.01)\,\text{bar}$$

and the desired pressure (1 bar) could not be reached with a good precision. This does not mean we could have neglected the pressure equilibration, since the initial pressure was at almost 40 000 bar.

These large fluctuations are most probably the result of the rather small number of atoms simulated (the $4 \times 4 \times 4$-supercell). To estimate the effect that these pressure fluctuations have on later calculations, which do not explicitly depend on the pressure, the change in volume due to these pressure fluctuations can be estimated. Using the isothermal compressibility κ_T, the relative change in volume is

$$\frac{\Delta V}{V} = -\kappa_T \Delta P \approx 0.0017 \tag{28}$$

where ΔP is taken to be the standard deviation of the achieved pressure. Therefore, the impact on further calculations is expected to be reasonably small.

The lattice constant after the equilibration is

$$a_0 = 4.0907\,\text{Å}\,,$$

which shows a reasonably good correspondence to the experimentally measured lattice constant of aluminium at atmospheric pressure and $T \approx 300\,\text{K}$ of 4.046 Å [8]. The simulated lattice constant is larger than the experimental value, however the temperature is also $\sim 500\,\text{K}$ higher in the simulation and therefore a larger lattice constant is expected.

At a temperature of around 500 °C in atmospheric pressure, aluminium is in a solid state. To check if the simulated system is also in a solid state, the time evolution of the positions of a few atoms are shown in figure 7. The atoms are staying at approximately their initial positions $\vec{r}(t = 0)$ and are not diffusing. This indicates that the system is indeed in a solid state.

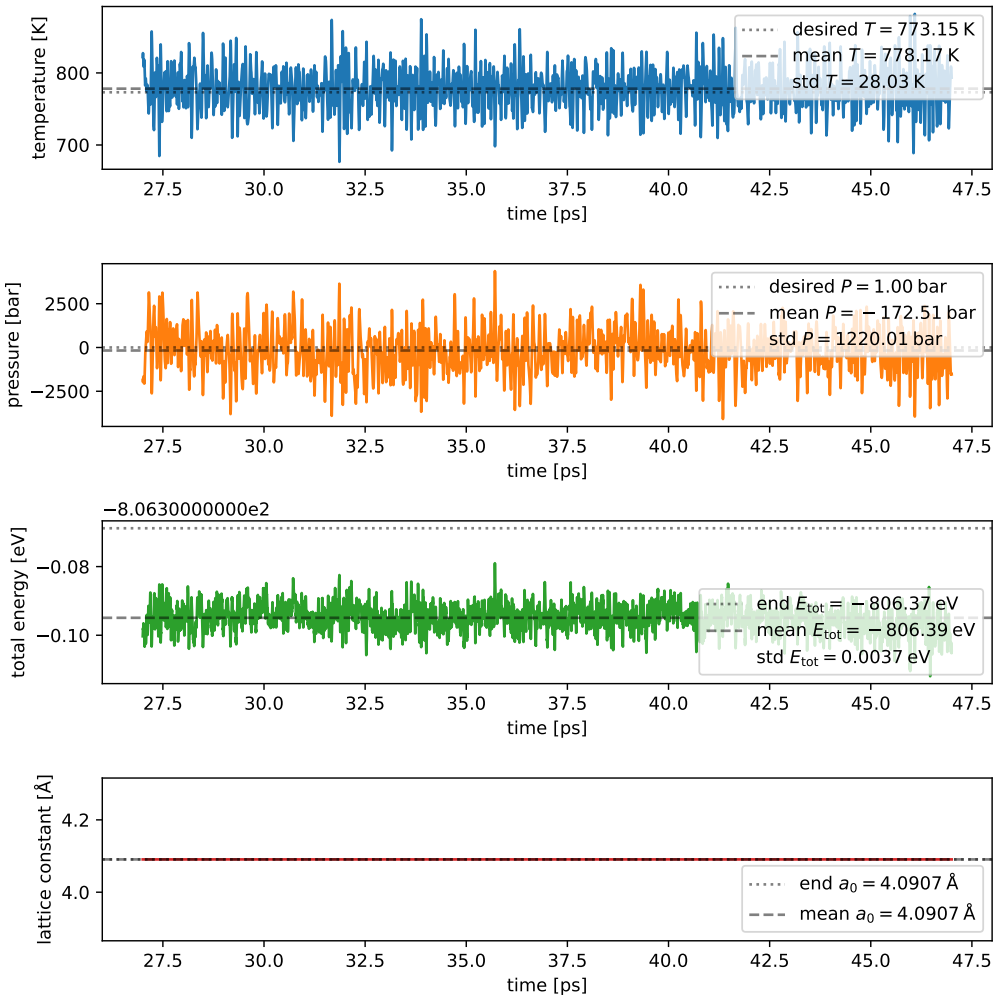

Figure 6: Time evolution of the instantaneous temperature, instantaneous pressure, total energy and lattice constant after the equilibration routine.

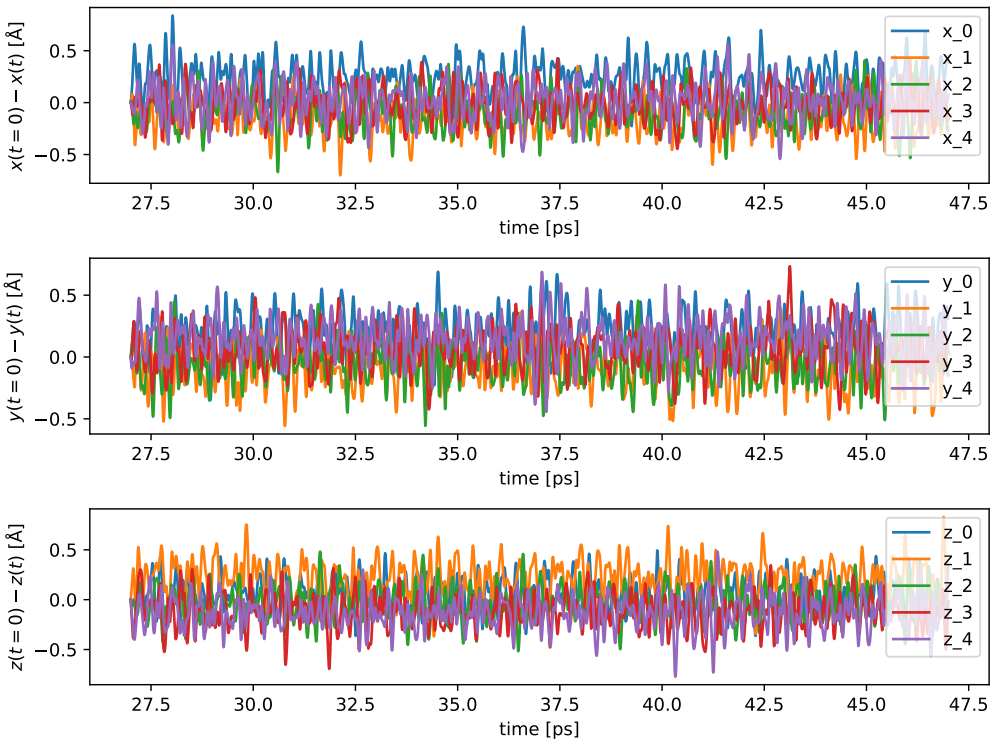

Figure 7: Time evolution of the relative positions (to the initial positions) of 5 particles in solid aluminium.

4 Task 4 2/2

4.1 Objective and Procedure

In this task, aluminium is simulated in a liquid state. For this, almost the same procedure is used as in task 3. However, the desired temperature is now $T = 700\,°\mathrm{C} = 973.15\,\mathrm{K}$ at a pressure $P = 1\,\mathrm{bar}$. The experimentally found melting point of aluminium at atmospheric pressure is $660.32\,°\mathrm{C}$ [3]. Therefore, it is expected to be in the fluid state at the desired temperature, if the simulation is reasonably close to reality.

However, in order to melt the system, it is recommended to first increase the temperature a good amount above the melting point and then cool it down (with the desired pressure being conserved). Therefore, the procedure chosen here is to equilibrate the temperature to $T = 1500\,°\mathrm{C}$, then decrease the pressure to $P = 1\,\mathrm{bar}$ at constant temperature and finally decrease the temperature to $T = 700\,°\mathrm{C}$ at constant pressure. The same $\delta t, \tau_T$ and τ_P as in task 3 are used.

4.2 Results and discussion

The equilibration process is shown in figure 8. Like in task 3, the temperature, pressure, total energy and lattice constant converge towards a certain value, which suggests an equilibrium state. This is again verified by simulating the system afterwards without position and velocity scaling, which is shown in fig. 9. The achieved average temperature and pressure are

$$T = \langle \mathcal{T}(t) \rangle_{\mathrm{time}} = (980.43 \pm 35.53)\,\mathrm{K}$$
$$P = \langle \mathcal{P}(t) \rangle_{\mathrm{time}} = (-211.50 \pm 1899.58)\,\mathrm{bar}$$

and show again large fluctuations, especially in the pressure. Since we assume the same isothermal compressibility, the same argument as in task 3 holds, why the pressure fluctuations do not matter for further calculations that are not explicitly dependent on the pressure. Therefore, the equilibration is considered successful, since the temperature and pressure are close enough to the desired values ($T = 973.15\,\mathrm{K}, P = 1\,\mathrm{bar}$) for this study. The total energy shows not only fluctuations per time step, but also seems to oscillate over larger time scales. However, the relative change in the total energy is negligible, and it can be considered as being conserved.

Again, the lattice parameter after equilibration is much larger than before, and it is also much larger than in task 3. This is expected as the volume had to be increased, and a fluid is generally less dense than a solid (except for water). However, the concept of a lattice parameter is not really applicable for a fluid state, but it still corresponds to the physical volume of the simulation cell.

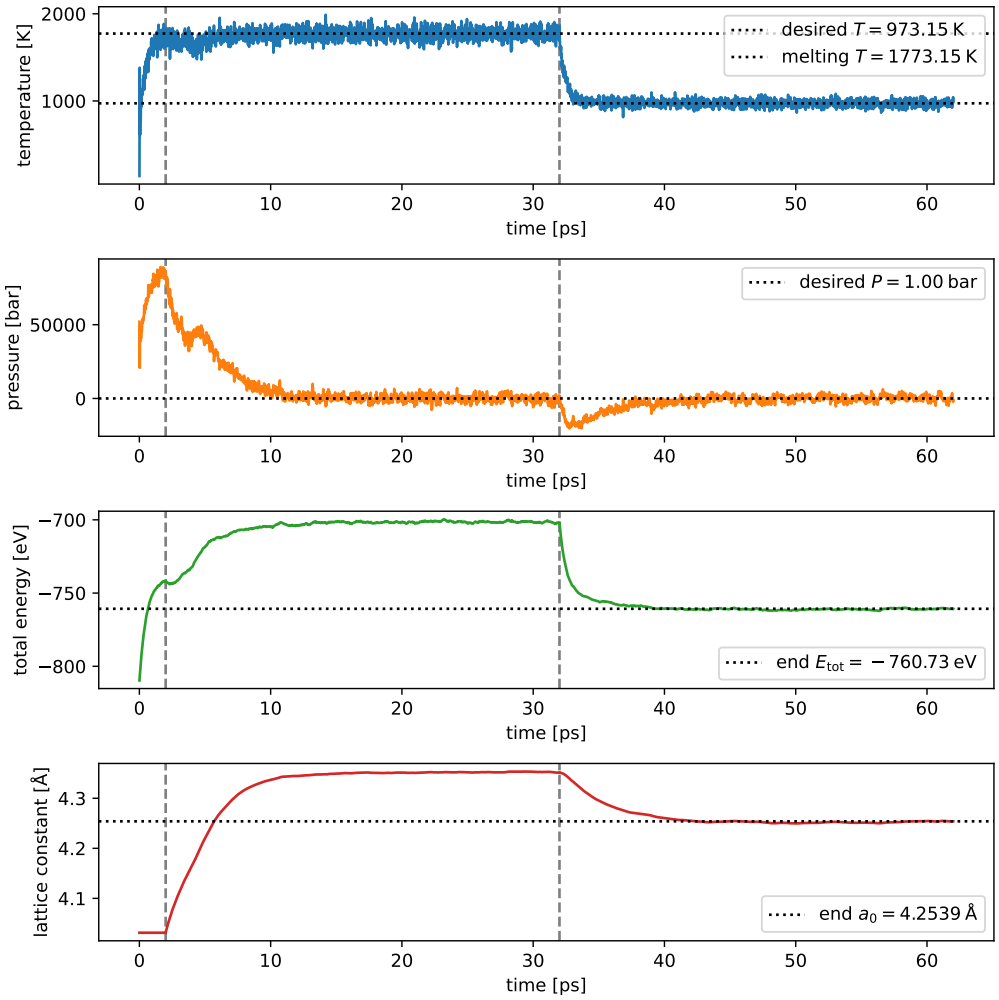

Figure 8: Time evolution of the instantaneous temperature, instantaneous pressure, total energy and lattice constant during the equilibration routine.

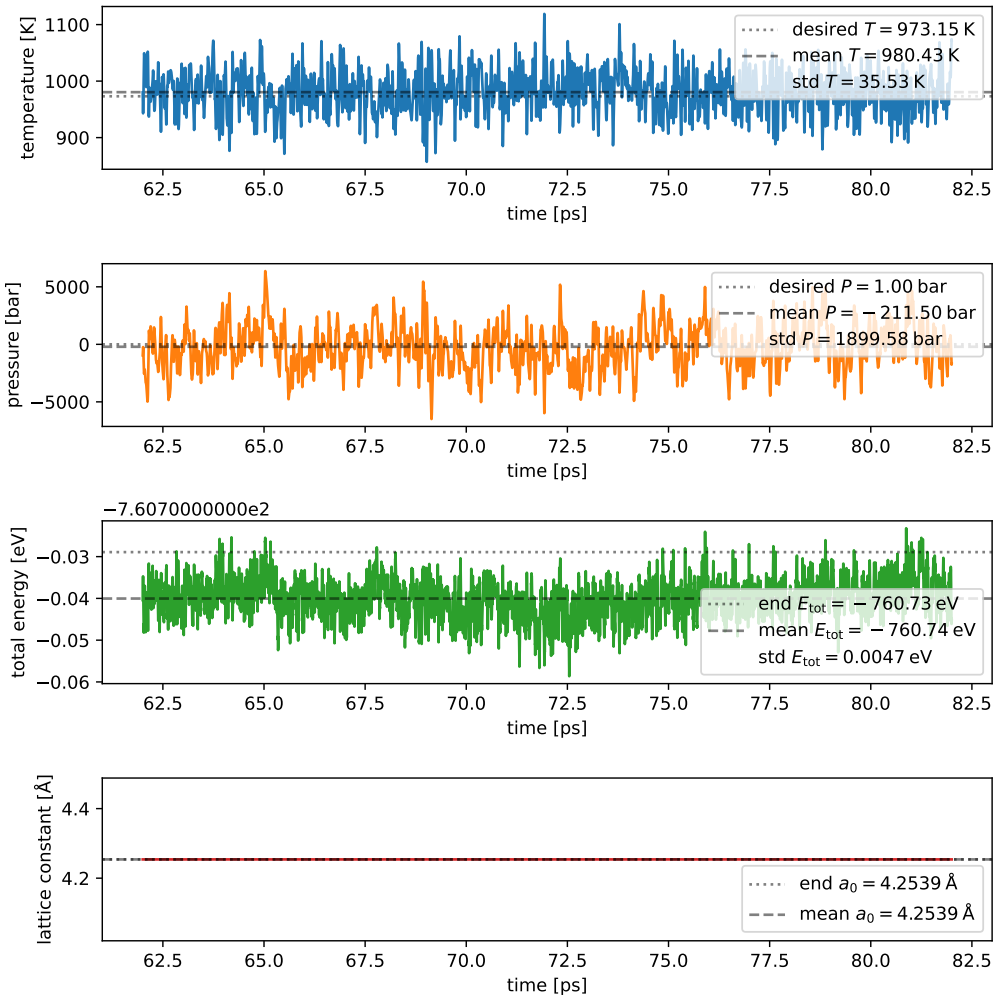

Figure 9: Time evolution of the instantaneous temperature, instantaneous pressure, total energy and lattice constant after the equilibration routine.

To evaluate if the system is in a liquid or solid state, the positions of a few atoms over time are shown in figure 10. The atoms clearly diffuse and tend to wander off in a specific direction. Therefore, it is concluded that the system is indeed in a liquid state.

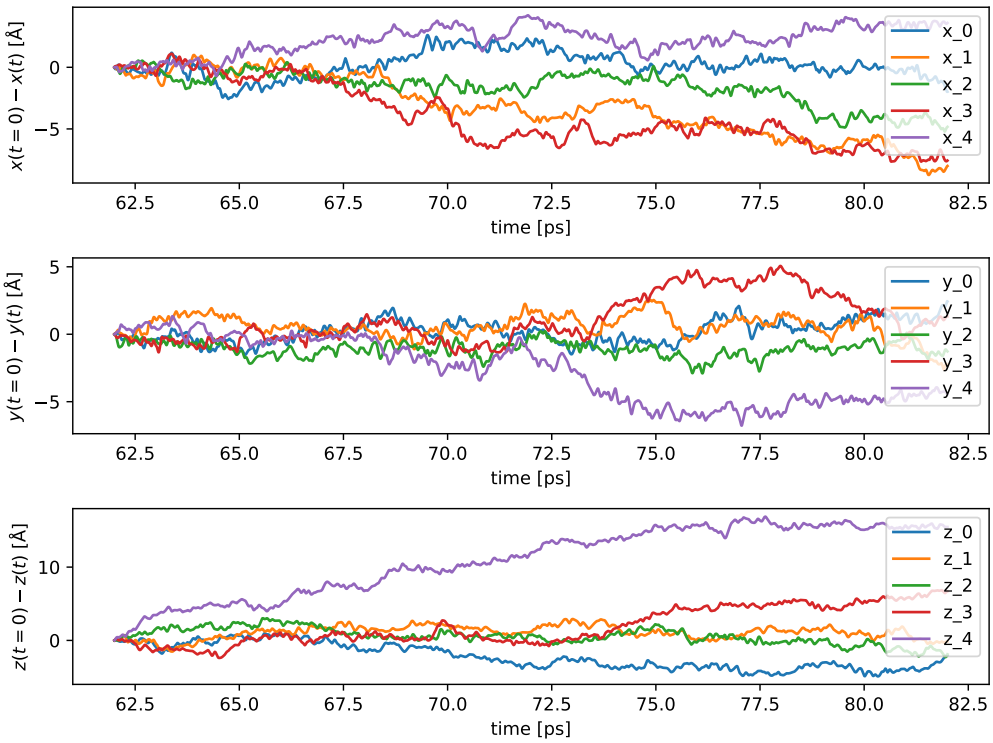

Figure 10: Time evolution of the relative positions (to the initial positions) of 5 particles in liquid aluminium.

5 Task 5 2/2

The simulations performed in task 3 (solid state) and task 4 (liquid) state are now further analyzed to determine the heat capacity C_V of the system. The important measure for this are the fluctuations in the energy and according to the lecture notes the heat capacity can be calculated as

$$C_V(E_{\text{kin}}) = \frac{3Nk_B}{2}\left[1 - \frac{2}{3Nk_B^2T^2}\left\langle(\delta E_{\text{kin}})^2\right\rangle\right]^{-1} \quad \text{or} \tag{29}$$

$$C_V(E_{\text{pot}}) = \frac{3Nk_B}{2}\left[1 - \frac{2}{3Nk_B^2T^2}\left\langle(\delta E_{\text{pot}})^2\right\rangle\right]^{-1} \tag{30}$$

where N is the number of atoms, k_B is the Boltzmann constant, T is the time average of the temperature and $\left\langle(\delta E)^2\right\rangle$ is the variance of the energy (kinetic or potential). To be able to compare this with experimental results, the specific heat capacity

$$c_V = \frac{C_V}{N \cdot m_{\text{Al}}} \tag{31}$$

is also calculated.

The results and experimental values are listed in table 1. There is no significant difference in using eq. (29) or eq. (30). For the solid state, the specific heat found through simulation is close to the experimentally found specific heat. However, for the liquid state, the value differs by $\sim 0.2\,\text{Jg}^{-1}\text{K}^{-1}$.

Table 1: Results for the heat capacity of the solid ($T = (505.02 \pm 28.03)\,°C$) and liquid ($T = (707.28 \pm 35.53)\,°C$) simulations and the corresponding specific heat capacities. To compare the results, experimentally measured specific heats are also given.

State	$C_V(E_{kin})$ / eVK^{-1}	$C_V(E_{pot})$ / eVK^{-1}	$c_V(E_{kin})$ / $Jg^{-1}K^{-1}$	$c_V(E_{pot})$ / $Jg^{-1}K^{-1}$	c_V(exp) / $Jg^{-1}K^{-1}$
Solid	0.0659	0.0663	0.921	0.926	0.91[9]
Liquid	0.0667	0.0670	0.932	0.936	1.18[10]

6 Task 6 4/4

The atomic structure of a material is an important part of investigating its properties. Here, the radial distribution function $g(r)$ of the simulated liquid aluminium is calculated. It describes the probability of finding an atom at radius r around another atom (assuming an isotropic system). To estimate $g(r)$ using the simulation, a binning has to be introduced, where $< N(r_k) >$ counts how many atoms are on average in the radial interval $[(k-1)\Delta r, k\Delta r]$ with $r_k = (k - 1/2)\Delta r$ $(k = 1, 2, 3, ..., N_{bins})$. The average is performed over all 256 simulated atoms and 10000 time steps, and the distances $|r_{ij}|$ between atom i and atom j are calculated using the minimum image convention

$$r_{ij} = (r_i - r_j) - L \cdot \left[\frac{(r_i - r_j)}{L}\right] \tag{32}$$

to account for the periodic boundary conditions. $L = 4a_0$ is the length of the simulation cell, and [...] denotes the rounding operation to the nearest integer. Then the binned radial distribution function can be calculated as

$$g(r_k) = \frac{\langle N(r_k)\rangle}{N_{ideal}(r_k)} \tag{33}$$

where

$$N_{ideal}(r_k) = \frac{N-1}{V}\frac{4\pi}{3}(k^2 - 3k + 1)\Delta r \tag{34}$$

is the average number of atoms in the same radial interval given that the system is completely randomly distributed.

The calculated radial distribution function of the liquid simulation is shown in fig. 11 using the binning

$$N_{bins} = 300 \qquad r_{max} = \frac{L}{2} = 8.5079\,Å \qquad \Delta r = 0.0284\,Å\,. \tag{35}$$

Clearly the first peak is also the global maximum, which corresponds to the 12 nearest neighbors in the fcc structure, which seems to be still dominant. To further investigate this, the coordination number

$$I(r_m) = \frac{N}{V}\int_0^{r_m} g(r) 4\pi r^2 \mathrm{d}r \tag{36}$$

is calculated, where $r_m = 3.8427\,Å$ is the first local minimum of $g(r)$. The resulting coordination number

$$I(r_m) = 11.8119 \tag{37}$$

emphasizes that there are 12 nearest neighbors in the fcc structure, which is approximately still the case in the molten aluminium (after a few picoseconds).

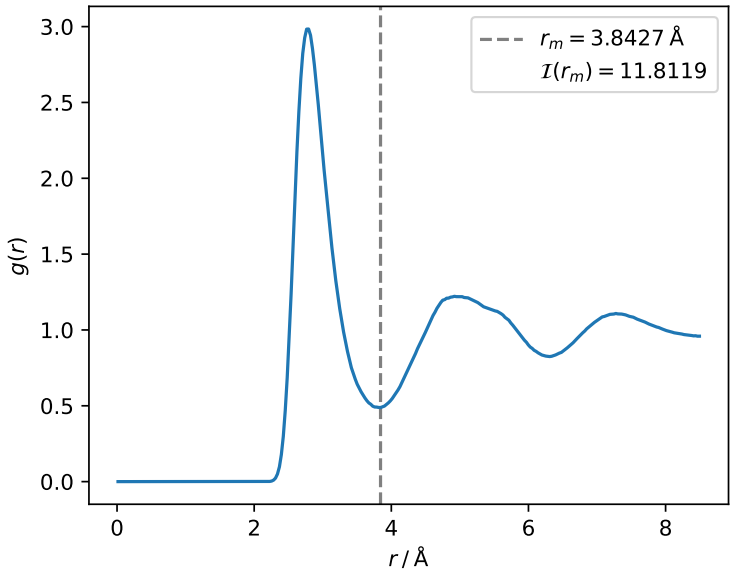

Figure 11: Radial distribution function calculated from the liquid aluminium simulation (task 4). Marked is also the first local minimum, which is used to calculate the coordination number $I(r_m)$.

7 Task 7 4/4

An essential tool in investigating materials are scattering (diffraction) experiments. Depending on the energy of the incident particles (usually photons) and the scattering angle, the intensity can both be calculated and measured. This is often used to infer information on the microscopic structure of a material. The intensity is proportional to the so-called static structure factor $S(\vec{q})$, where $\vec{q}$ is the wave vector of the outgoing particles, which assumes that the scattering is elastic (no momentum transfer) and that the incoming particles always approach from the same angle.

Here, the structure factor of the simulated liquid aluminium (task 4) is calculated through

$$S(\vec{q}) = \frac{1}{N}\left\langle\left(\sum_{i=1}^{N}\cos\left[\vec{q}\cdot\vec{r}_i(t)\right]\right)^2 + \left(\sum_{i=1}^{N}\sin\left[\vec{q}\cdot\vec{r}_i(t)\right]\right)^2\right\rangle_{\text{time}} \tag{38}$$

where $\vec{r}_i(t)$ is the position of atom i at time t. This is calculated for each

$$\vec{q}_{n_x,n_y,n_z} = \frac{2\pi}{L}(n_x, n_y, n_z)^T \quad \text{with } n_x, n_y, n_z = 0, 1, 2, ..., N_{\text{grid}} \tag{39}$$

in an equally spaced grid in the reciprocal space. Due to the computational cost of this calculation, it is calculated for every 100th time step (instead of in each of the 10000 time steps) with $N_{\text{grid}} = 25$.

Shown in fig. 12 is the calculated static structure factor averaged over the same magnitudes of $\vec{q}$. To reduce noise, the distances are binned (300 bins) and then the average (and standard deviation) of $S(q)$ is taken. In comparison to the experimentally measured structure factor (fig. 13) at $T = 703\,^\circ\text{C}$ of liquid aluminium, a quite similar structure factor is found in our simulation at $T = (707.28 \pm 35.53)\,^\circ\text{C}$. Especially when considering the small number of simulated unit cells and all the approximations done for calculating the potential.

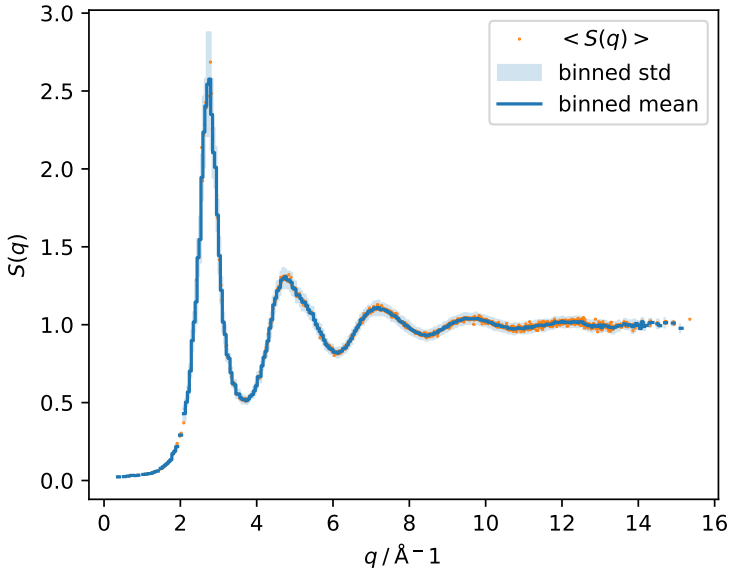

Figure 12: Static structure factor of the simulated liquid aluminium (task 4). Both the binned averaging (blue) and the averaging over same distances (orange) is shown.

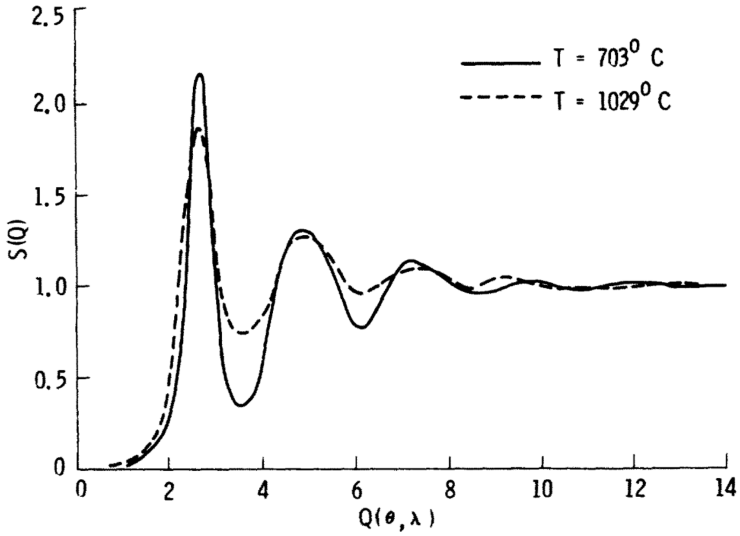

Figure 13: Experimentally measured structure factor of liquid aluminium using neutron diffraction. [11]

8 Concluding discussion

In tasks 1 to 4 of this assignment, it was shown that a simplified model of aluminium using only a $4 \times 4 \times 4$ supercell (periodic boundary conditions) can be simulated and results in reasonable physics. It was possible to simulate the system and conserve the energy during the simulation. Furthermore, the temperature and pressure of the system was adjusted using an equilibration technique that involves scaling the positions and velocities. Due to the small number of atoms that are simulated, the variation in the pressure is very large, and it could not be equilibriated to precisely 1 bar. Finally, it could be shown that the simulated aluminium crystal can behave both like a solid (task 3) and a liquid (task 4). The lattice constants are close to the experimentally measured values in the solid states in task 1 and 3. For the liquid state int task 4, the concept of a lattice parameter is not well-defined, but still yields a much larger lattice constant than in the solid state, which is expected.

In tasks 5 to 7 of this assignment, it was shown that the calculation of some static properties yields reasonable results and compares well to experimental values. The specific heat of the simulated solid aluminium is quite close to the experimentally found value, however for the liquid state a larger variation from the experimental value is found. The calculated radial distribution function has the expected shape and results in a reasonable coordination number. The static structure factor, closely resembles the experimentally found static structure factor. The differences to the experimental values can most likely be again explained by the small number of unit cells simulated.

Overall, it was shown that molecular dynamics simulations using the velocity Verlet algorithm can be used to simulate materials, and it is possible to estimate some physical properties of this material.

References

[1] M. Mantina, A. Chamberlin et al.: "Consistent van der Waals Radii for the Whole Main Group", The Journal of Physical Chemistry A (2009), 113 (19): 5806–5812. https://doi.org/10.1021/jp8111556

[2] A. K. Giri and G. B. Mitra: "Extrapolated values of lattice constants of some cubic metals at absolute zero", J. Phys. D: Appl. Phys. (1985), 18, L75, https://doi.org/10.1088/0022-3727/18/7/005.

[3] Wikipedia: "Aluminium", https://en.wikipedia.org/wiki/Aluminium, visited on 25/11/2022.

[4] "Molecular Dynamics: Velocity Verlet (data page)", https://python.plainenglish.io/molecular-dynamics-velocity-verlet-integration-with-python-5ae66b63a8fd, visited 24/11/2022.

[5] G. Wahnström: Molecular Dynamics, 27/10/2021.

[6] C. Kittel: "Introduction to Solid State Physics", 8th edition, p. 52, Hoboken, NJ: John Wiley & Sons, Inc, 2005.

[7] Wikipedia: "Verlet Integration (data page)", https://en.wikipedia.org/wiki/Verlet_integration, visited on 24/11/2022.

[8] Wheeler Davey: "Precision Measurements of the Lattice Constants of Twelve Common Metals", Physical Review (1925), 25 (6): 753–761. https://doi.org/10.1103/PhysRev.25.753

[9] Engineering ToolBox: "Metals - Specific Heats." (2003), https://www.engineeringtoolbox.com/specific-heat-metals-d_152.html, visited on 08/12/2022.

[10] Engineering ToolBox: "Metals - Boiling Points and Specific Heat." (2014), https://www.engineeringtoolbox.com/liquid-metal-boiling-points-specific-heat-d_1893.html, visited on 08/12/2022.

[11] J. M. Stallard and C. M. Davis, Jr.: "Liquid-Aluminum Structure Factor by Neutron Diffraction", Phys. Rev. A (1973) 8, 368, https://doi.org/10.1103/PhysRevA.8.368.

A Source Code

Included in this appendix is all the code that I wrote myself. The entire project is attached as a zip file, where also the code is included that was already provided.

A.1 Main code for all calculations/simulations: run.c

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <gsl/gsl_rng.h>
#include "lattice.h"
#include "potential.h"
#include "tools.h"

// unit conversion factors
const double eV = 1.602176634e-19; // J
const double m_asu = 9649; // u
const double m_asu_kg = 1.6028e-23; // kg
const double GPa = 1e-21 / eV; // eV/Å^3
const double bar = 1e-4 * GPa; // eV/Å^3
const double celsius = 273.15; // K

// Simulation/Physics constants
const double m = 26.98153853 / m_asu; // m_asu (https://physics.nist.gov/cgi-bin/Compositions/stand_alone.pl?ele=Al)
const double k_B = 8.617333262e-5; // eV/K (https://en.wikipedia.org/wiki/Boltzmann_constant#Value_in_different_units)
const double kappa_T = 0.01385 / GPa; // Å^3/eV https://www.knowledgedoor.com/2/elements_handbook/isothermal_compressibility.html
const int n_cells = 4; // number of unit cells
const int n_atoms = 4*n_cells*n_cells*n_cells;


void task1(){
    double pos[n_atoms][3];

    double a0_min = 3.98; // Å
    double a0_max = 4.08; // Å
    int n_a0 = 1000; // how many different a0 should be used?
    double da0 = (a0_max-a0_min)/(n_a0-1); // spacing of the different a0

    // prepare the output file
    FILE* file = fopen("data/H1_1.csv","w");
    fprintf(file, "# a0[Å], E_pot[eV]\n");

    for(int i=0;i<n_a0;i++){
        // calculate the cell lenghths
        double a0 = a0_min + i*da0; // Å
        double L_box = n_cells*a0;

        init_fcc(pos, n_cells, a0);
        double E_pot = get_energy_AL(pos, L_box, n_atoms);

        fprintf(file, "%.8f, %.10f\n", a0, E_pot);
    }
    fclose(file);
}

void initialize_lattice(double pos[][3], double vel[][3], double a0){
    init_matrix_stack(n_atoms,3,pos,0);
    init_matrix_stack(n_atoms,3,vel,0);

    init_fcc(pos, n_cells, a0);

    // initialize random number generator
    gsl_rng_env_setup();
    const gsl_rng_type* T = gsl_rng_default;
    gsl_rng* rng = gsl_rng_alloc(T);

    double eps_min = -a0*0.065;
    double eps_max = a0*0.065;

    // introduce deviations
    for(int i=0;i<n_atoms;i++){
        pos[i][0] += gsl_rng_uniform(rng)*(eps_max-eps_min) + eps_min;
        pos[i][1] += gsl_rng_uniform(rng)*(eps_max-eps_min) + eps_min;
        pos[i][2] += gsl_rng_uniform(rng)*(eps_max-eps_min) + eps_min;
    }
    gsl_rng_free(rng);
}

void velocity_verlet_timestep(double* t, double dt, double a0,
                              double pos[][3], double vel[][3], double F[][3]) {
    double L_box = n_cells*a0;
```

```
76      /* v(t+dt/2) */
77      for (int j = 0; j < n_atoms; j++) {
78          vel[j][0] += dt * 0.5 * F[j][0]/m;
79          vel[j][1] += dt * 0.5 * F[j][1]/m;
80          vel[j][2] += dt * 0.5 * F[j][2]/m;
81      }
82
83      /* q(t+dt) */
84      for (int j = 0; j < n_atoms; j++) {
85          pos[j][0] += dt * vel[j][0];
86          pos[j][1] += dt * vel[j][1];
87          pos[j][2] += dt * vel[j][2];
88      }
89
90      /* F(t+dt) */
91      get_forces_AL(F,pos,L_box,n_atoms);
92
93      /* v(t+dt) */
94      for (int j = 0; j < n_atoms; j++) {
95          vel[j][0] += dt * 0.5 * F[j][0]/m;
96          vel[j][1] += dt * 0.5 * F[j][1]/m;
97          vel[j][2] += dt * 0.5 * F[j][2]/m;
98      }
99
100     //increase the time
101     (*t) += dt;
102 }
103
104 void simulate_and_save_data(double pos[][3], double vel[][3], double* t,
105                             char* filename,
106                             int n_timesteps, double dt, double* a0,
107                             double temp_scaling_time, double T_desired,
108                             double pressure_scaling_time, double P_desired,
109                             int n_save_positions){
110     // if the scaling times are <= 0 then no corresponding scaling happens (no equilibration)
111
112     double run_time = (n_timesteps+1)*dt;
113
114     // prepare output file
115     FILE* file = fopen(filename, "w");
116     fprintf(file, "# {\"dt\": %.5e, \"n_timesteps\": %i, \"run_time\": %.5e, \n",dt, n_timesteps, run_time);
117     fprintf(file, "# \"temp_scaling_time\": %.5e, \"T_desired\": %.5e, \n", temp_scaling_time, T_desired);
118     fprintf(file, "# \"pressure_scaling_time\": %.5e, \"P_desired\": %.5e, \n", pressure_scaling_time, P_desired/↩
            bar);
119     fprintf(file, "# \"n_atoms\": %i, \"n_save_positions\": %i}\n", n_atoms, n_save_positions);
120
121     // table header
122     fprintf(file, "# t[ps], E_pot[eV], E_kin[eV], T[K], P[bar], a0[Å]");
123     for(int j=0;j<n_save_positions;j++){
124         fprintf(file, ", x_1[Å]");
125         fprintf(file, ", y_1[Å]");
126         fprintf(file, ", z_1[Å]");
127     }
128     fprintf(file, "\n");
129
130     // velocity verlet:
131     printf("Simulate \"%s\"...\n", filename);
132     printf("dt = %.5e ; n_timesteps = %i ; run_time = %.5f\n", dt, n_timesteps, run_time);
133
134     double a0_ = *a0;
135
136     double L_box = n_cells*a0_;
137
138     // prepare all needed variables for inside the algorithm
139     int percent = -1; // for displaying progress
140     double E_pot = 0; // eV
141     double E_kin = 0; // eV
142     double T = 0; // K
143     double P = 0; // eV/Å^3
144     double t_= *t; // ps
145     double F[n_atoms][3]; //forces eV/Å
146     get_forces_AL(F,pos,L_box,n_atoms);
147
148     // evolve the system over time
149     for(int i=1;i<n_timesteps+1;i++){
150         // show progress
151         if((i+1)*100/(n_timesteps+1) != percent) {
152             percent = (i+1)*100/(n_timesteps+1);
153             printf("\33[2K\r%i %% ; timestep = %i", percent, i);
154             fflush(stdout);
155         }
156
157         velocity_verlet_timestep(&t_, dt, a0_, pos, vel, F);
158
159         // calculate the energies
160         E_pot = get_energy_AL(pos, L_box, n_atoms);
161         E_kin = 0;
162         for(int j=0;j<n_atoms;j++){
163             E_kin += 0.5*m*vel[j][0]*vel[j][0];
164             E_kin += 0.5*m*vel[j][1]*vel[j][1];
```

```
            E_kin += 0.5*m*vel[j][2]*vel[j][2];
        }

        // calculate the instantaneaous properties (temperature, pressure)
        T = (2/(3*n_atoms*k_B)) * E_kin;
        P = (n_atoms * k_B * T + get_virial_AL(pos, L_box, n_atoms)) / (L_box*L_box*L_box);

        if(temp_scaling_time > 0){
            // equilibration scaling(temperature)
            double alpha_T_sqrt = sqrt(1 + 2*dt/temp_scaling_time * (T_desired - T)/T);
            for(int j=0;j<n_atoms;j++){
                vel[j][0] *= alpha_T_sqrt;
                vel[j][1] *= alpha_T_sqrt;
                vel[j][2] *= alpha_T_sqrt;
            }
        }

        if(pressure_scaling_time > 0){
            // equilibration scaling(pressure)
            double alpha_P_cbrt = cbrt(1 - kappa_T*dt/pressure_scaling_time * (P_desired - P));
            for(int j=0;j<n_atoms;j++){
                pos[j][0] *= alpha_P_cbrt;
                pos[j][1] *= alpha_P_cbrt;
                pos[j][2] *= alpha_P_cbrt;
            }
            L_box *= alpha_P_cbrt;
            a0_ *= alpha_P_cbrt;
        }

        P = P / bar; // GPa

        // save the data
        fprintf(file, "%.10f, %.10e, %.10e, %.10e, %.10e, %.10e", t_, E_pot, E_kin, T, P, a0_);
        for(int j=0;j<n_save_positions;j++){
            fprintf(file, ", %.5e", pos[j][0]);
            fprintf(file, ", %.5e", pos[j][1]);
            fprintf(file, ", %.5e", pos[j][2]);
        }
        fprintf(file, "\n");
    }
    printf("\n");
    // update the variables outside this function
    *a0 = a0_;
    *t = t_;

    fclose(file);
}

void save_system_state(char* file_path, int n_atoms, double pos[n_atoms][3], double vel[n_atoms][3], double a0, ↩
     double t, double T, double P) {
    FILE* file = fopen(file_path, "w");
    // Header
    fprintf(file, "# t[ps] = %.10f\n", t);
    fprintf(file, "# a0[Å] = %.10f\n", a0);
    fprintf(file, "# T[K] = %.10f\n", T);
    fprintf(file, "# P[bar] = %.10f\n", P/bar);
    fprintf(file, "# n_atoms = %i\n", n_atoms);
    fprintf(file, "# pos[0], pos[1], pos[2], vel[0], vel[1], vel[2]\n");
    for (int i=0; i<n_atoms; i++) {
        fprintf(file, "%.10f, %.10f, %.10f, %.10f, %.10f, %.10f\n",
            pos[i][0], pos[i][1], pos[i][2], vel[i][0], vel[i][1], vel[i][2]);
    }
    fclose(file);
}

void load_system_state(char* file_path, int n_atoms, double pos[n_atoms][3], double vel[n_atoms][3], double* a0, ↩
     double* t, double* T, double* P) {
    FILE* file = fopen(file_path, "r");
    char buf[255];
    int n_atoms_test = 0;
    int res;

    res = fscanf(file, "# t[ps] = %lf\n", t);
    res = fscanf(file, "# a0[Å] = %lf\n", a0);
    res = fscanf(file, "# T[K] = %lf\n", T);
    res = fscanf(file, "# P[bar] = %lf\n", P);
    (*P) *= bar;
    res = fscanf(file, "# n_atoms = %i\n", &n_atoms_test);
    if (n_atoms_test != n_atoms) {
        perror("ERROR: n_atoms_test != n_atoms");
        exit(1);
    }
    res = fscanf(file, "# pos[0], pos[1], pos[2], vel[0], vel[1], vel[2]\n");
    for (int i=0; i<n_atoms; i++) {
        res = fscanf(file, "%lf, %lf, %lf, %lf, %lf, %lf\n",
            &pos[i][0], &pos[i][1], &pos[i][2], &vel[i][0], &vel[i][1], &vel[i][2]);
    }
    fclose(file);
}

```

```
253 void task2(double dt, char* filename){
254     double pos[n_atoms][3]; //positions
255     double vel[n_atoms][3]; //velocities
256 
257     double a0 = 4.03139; // Å
258 
259     double t_max = 15;
260     int n_timesteps = t_max/dt;
261 
262     double t = 0;
263 
264     initialize_lattice(pos, vel, a0);
265 
266     simulate_and_save_data(pos, vel, &t, filename, n_timesteps, dt, &a0, 0, 0, 0, 0, 0);
267 }
268 
269 void task3(){
270     double pos[n_atoms][3]; //positions
271     double vel[n_atoms][3]; //velocities
272 
273     double a0 = 4.03139; // Å
274     double dt = 5e-3; //ps
275     double t = 0;
276 
277     double tau_T = 100*dt;
278     double T_desired = 500 + celsius; // K
279     int n_timesteps_temp_scaling = (6*tau_T)/dt;
280 
281     double tau_P = 3*tau_T;
282     double P_desired = 1 * bar; // eV/Å^3
283     int n_timesteps_pressure_scaling = (16*tau_P)/dt;
284 
285     double dt_simulation = 2e-3; //ps
286     int n_timesteps_simulation = 10000;
287 
288     printf("P_desired = %.5e eV/Å^3\n", P_desired);
289     printf("kappa_T = %.5e eV/Å^3\n", kappa_T);
290 
291     initialize_lattice(pos, vel, a0);
292 
293     // temperature equilibriation
294     simulate_and_save_data(pos, vel, &t,
295                            "data/H1_3_temp_scaling.csv",
296                            n_timesteps_temp_scaling, dt, &a0,
297                            tau_T, T_desired, 0, 0, 0);
298     // pressure equilibriation
299     simulate_and_save_data(pos, vel, &t,
300                            "data/H1_3_pressure_scaling.csv",
301                            n_timesteps_pressure_scaling, dt, &a0,
302                            tau_T, T_desired,
303                            tau_P, P_desired, 0);
304 
305     // simulate
306     simulate_and_save_data(pos, vel, &t,
307                            "data/H1_3_after_scaling.csv",
308                            n_timesteps_simulation, dt_simulation, &a0,
309                            0,0,0,0,
310                            5);
311 
312     save_system_state("data/H1_solid_state.csv", n_atoms, pos, vel, a0, t, T_desired, P_desired);
313 }
314 
315 void task4(){
316     double pos[n_atoms][3]; //positions
317     double vel[n_atoms][3]; //velocities
318 
319     double a0 = 4.03139; // Å
320     double dt = 5e-3; //ps
321     double t = 0;
322 
323     double tau_T = 100*dt;
324     double tau_P = 3*tau_T;
325 
326     double T_desired_melting = 1500 + celsius; // K
327     double P_desired = 1 * bar; // eV/Å^3
328     double T_desired = 700 + celsius; // K
329 
330 
331     int n_timesteps_temp_scaling = (4*tau_T)/dt;
332     int n_timesteps_pressure_scaling = (20*tau_P)/dt;
333     int n_timesteps_temp_decreasing = (20*tau_P)/dt;
334 
335     double dt_simulation = 2e-3; //ps
336     int n_timesteps_simulation = 10000;
337 
338     printf("P_desired = %.5e eV/Å^3\n", P_desired);
339     printf("kappa_T = %.5e eV/Å^3\n", kappa_T);
340 
341     initialize_lattice(pos, vel, a0);
342 
```

```
343     // temperature equilibriation
344     simulate_and_save_data(pos, vel, &t,
345                            "data/H1_4_temp_scaling.csv",
346                            n_timesteps_temp_scaling, dt, &a0,
347                            tau_T, T_desired_melting, 0, 0, 0);
348     // pressure equilibriation while melting
349     simulate_and_save_data(pos, vel, &t,
350                            "data/H1_4_pressure_scaling.csv",
351                            n_timesteps_pressure_scaling, dt, &a0,
352                            tau_T, T_desired_melting,
353                            tau_P, P_desired, 0);
354
355     // temperature equilibriation to lower temperature while it is melted
356     simulate_and_save_data(pos, vel, &t,
357                            "data/H1_4_temp_decreasing.csv",
358                            n_timesteps_temp_decreasing, dt, &a0,
359                            tau_T, T_desired,
360                            tau_P, P_desired, 0);
361
362     // simulate
363     simulate_and_save_data(pos, vel, &t,
364                            "data/H1_4_after_scaling.csv",
365                            n_timesteps_simulation, dt_simulation, &a0,
366                            0,0,0,0,
367                            5);
368
369     save_system_state("data/H1_liquid_state.csv", n_atoms, pos, vel, a0, t, T_desired, P_desired);
370 }
371
372 void task6() {
373     double pos[n_atoms][3]; //positions
374     double vel[n_atoms][3]; //velocities
375     double F[n_atoms][3]; //forces
376     double a0, t, T, P;
377
378     load_system_state("data/H1_liquid_state.csv", n_atoms, pos, vel, &a0, &t, &T, &P);
379
380     double dt = 5e-3; //ps
381     int n_timesteps = 10000;
382     double run_time = (n_timesteps+1)*dt;
383
384     double L_box = n_cells*a0;
385     int percent = -1; // for displaying progress
386
387     // prepare the histogram
388     int n_bins = 300;
389     double r_max = L_box/2;
390     double dr = r_max/n_bins;
391     double N_r[n_bins];
392     for (int i_bin=0; i_bin<n_bins; i_bin++) {
393         N_r[i_bin] = 0;
394     }
395
396     printf("Task 6: Simulate and calculate the pair distance histogram...\n");
397     printf("dt = %.5e ; n_timesteps = %i ; run_time = %.5f\n", dt, n_timesteps, run_time);
398
399     get_forces_AL(F,pos,L_box,n_atoms);
400
401     for (int i_step=0; i_step<n_timesteps; i_step++) {
402         // show progress
403         if((i_step+1)*100/(n_timesteps+1) != percent) {
404             percent = (i_step+1)*100/(n_timesteps+1);
405             printf("\33[2K\r%i %% ; timestep = %i", percent, i_step);
406             fflush(stdout);
407         }
408
409         velocity_verlet_timestep(&t, dt, a0, pos, vel, F);
410
411         // calculate the pair distances for the <N(r)> histogram
412         for (int k=0; k<n_atoms; k++) {
413         for (int l=k+1; l<n_atoms; l++) {
414             // distance in the minimum image convention
415             double dx[3];
416             for (int i_x=0; i_x<3; i_x++) {
417                 dx[i_x] = pos[k][i_x] - pos[l][i_x];
418                 dx[i_x] = dx[i_x] - L_box*round(dx[i_x]/L_box);
419             }
420             double dist = vector_norm(dx, 3);
421
422             // add it to the right bin (twice to accommodate for both N(r))
423             if (dist < r_max) {
424                 int i_bin = (int) (dist/dr);
425                 N_r[i_bin] += 2;
426             }
427         }
428         }
429     }
430     printf("\n");
431
432     // take the average instead of the absolute count
433     for (int i_bin=0; i_bin<n_bins; i_bin++) {
```

```
        N_r[i_bin] /= (double) n_atoms*n_timesteps;
    }

    // write to a file
    FILE* file = fopen("data/H1_6.csv", "w");
    fprintf(file, "# {\"n_atoms\": %i, \"L_box[Å]\": %.10f}\n", n_atoms, L_box);
    fprintf(file, "# r[Å], <N(r)>\n");
    for (int i_bin=0; i_bin<n_bins; i_bin++) {
        fprintf(file, "%.10f, %.10f\n", (i_bin+0.5)*dr, N_r[i_bin]);
    }
    fclose(file);
}



void task7() {
    double pos[n_atoms][3]; //positions
    double vel[n_atoms][3]; //velocities
    double F[n_atoms][3]; //forces
    double a0, t, T, P;

    load_system_state("data/H1_liquid_state.csv", n_atoms, pos, vel, &a0, &t, &T, &P);

    double dt = 5e-3; //ps
    int n_timesteps = 10000;
    int n_skip_timesteps = 9;
    double run_time = (n_timesteps+1)*dt;

    double L_box = n_cells*a0;
    int percent = -1; // for displaying progress

    // prepare the q grid
    int n_grid = 25; // number of grid points per dimension
    int n_q = n_grid*n_grid*n_grid; // number of grid points in total
    double** q = create_2D_array(n_grid*n_grid*n_grid, 3);
    double* S_q = (double*)calloc(n_grid*n_grid*n_grid, sizeof(double));
    int i_q = 0;
    for (int n_x=0; n_x<n_grid; n_x++) {
    for (int n_y=0; n_y<n_grid; n_y++) {
    for (int n_z=0; n_z<n_grid; n_z++) {
        q[i_q][0] = 2*M_PI/L_box * n_x;
        q[i_q][1] = 2*M_PI/L_box * n_y;
        q[i_q][2] = 2*M_PI/L_box * n_z;
        i_q++;
    }
    }
    }

    printf("Task 5: Simulate and calculate the structure factors...\n");
    printf("dt = %.5e ; n_timesteps = %i ; run_time = %.5f\n", dt, n_timesteps, run_time);

    get_forces_AL(F,pos,L_box,n_atoms);

    int skip_counter = n_skip_timesteps;
    int used_timestep_counter = 0;

    for (int i_step=0; i_step<n_timesteps; i_step++) {
        // show progress
        if((i_step+1)*100/(n_timesteps+1) != percent) {
            percent = (i_step+1)*100/(n_timesteps+1);
            printf("\33[2K\r%i %% ; timestep = %i", percent, i_step);
            fflush(stdout);
        }

        velocity_verlet_timestep(&t, dt, a0, pos, vel, F);

        if (skip_counter<n_skip_timesteps) {
            skip_counter++;
            continue;
        }
        skip_counter = 0;
        used_timestep_counter++;

        // calculate the structure factors
        for (int i_q=0; i_q<n_q; i_q++) {
            double cos_sum = 0;
            double sin_sum = 0;
            for (int i_atom=0; i_atom<n_atoms; i_atom++) {
                // minimum image conventiion position of the atom
                double temp_pos[3];
                temp_pos[0] = pos[i_atom][0] - L_box*round(pos[i_atom][0]/L_box);
                temp_pos[1] = pos[i_atom][1] - L_box*round(pos[i_atom][1]/L_box);
                temp_pos[2] = pos[i_atom][2] - L_box*round(pos[i_atom][2]/L_box);

                double q_dot_r = dot_product(q[i_q],temp_pos, 3);
                cos_sum += cos(q_dot_r);
                sin_sum += sin(q_dot_r);
            }
            S_q[i_q] += (cos_sum*cos_sum + sin_sum*sin_sum)/n_atoms;
        }
```

```
524     }
525     printf("\n");
526
527     // take the average instead of the sum
528     for (int i_q=0; i_q<n_q; i_q++) {
529         S_q[i_q] /= used_timestep_counter;
530     }
531
532     FILE* file = fopen("data/H1_7.csv", "w");
533     fprintf(file, "# {\"n_timesteps\": %i, \"n_skip_timesteps\": %i, \"used_timesteps\": %i, \"n_grid\": %i}\n", ↩
            n_timesteps, n_skip_timesteps, used_timestep_counter, n_grid);
534     fprintf(file, "# q_x[Å^-1], q_x[Å^-1], q_x[Å^-1], S(q)\n");
535     for (int i_q=0; i_q<n_q; i_q++) {
536         double q_norm = vector_norm(q[i_q], 3);
537         fprintf(file, "%.10f, %.10f, %.10f, %.10f, %.10f\n", q[i_q][0], q[i_q][1], q[i_q][2], q_norm, S_q[i_q]);
538     }
539     fclose(file);
540
541     // free stuff
542     destroy_2D_array(q);
543     free(S_q);
544 }
545
546 int
547 run(
548     int argc,
549     char *argv[]
550    )
551 {
552     // Write your code here
553     // This makes it possible to test
554     // 100% of you code
555
556     // because every task takes some time, all tasks that have already been calculated are commented out
557     task1();
558     task2(5e-3, "data/H1_2_small_enough.csv");
559     task2(50e-3, "data/H1_2_far_too_large.csv");
560     task2(15e-3, "data/H1_2_little_too_large.csv");
561     task3();
562     task4();
563     task6();
564     task7();
565
566     return 0;
567 }
```

A.2 Utility functions: tools.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5  #include "tools.h"
6
7  void constant_multiplication(double* res,
8                               double* v1,
9                               double a1,
10                              unsigned int len)
11 {
12     for(int i=0;i<len;i++){
13         res[i] = a1*v1[i];
14     }
15 }
16
17 void
18 elementwise_addition(
19             double *res,
20             double *v1,
21             double *v2,
22             unsigned int len
23                 )
24 {
25     for(int i=0;i<len;i++){
26         res[i] = v1[i] + v2[i];
27     }
28 }
29
30 void
31 elementwise_multiplication(
32                double *res,
33                double *v1,
34                double *v2,
35                unsigned int len
36                       )
37 {
38     for(int i=0;i<len;i++){
```

```
39          res[i] = v1[i] * v2[i];
40      }
41  }
42
43  double
44  dot_product(
45          double *v1,
46          double *v2,
47          unsigned int len
48         )
49  {
50      double res = 0;
51      for(int i=0;i<len;i++){
52          res += v1[i] * v2[i];
53      }
54      return res;
55  }
56
57  double**
58  create_2D_array(
59          unsigned int nrows,
60          unsigned int ncols
61             )
62  {
63      // allocate 1D array of doubles containing the whole matrix
64      double* linear_array = (double*)malloc(nrows*ncols*sizeof(double));
65      // allocate 1D array of pointers to doubles containing the pointers to each row starting point
66      double** array = (double**)malloc(nrows*sizeof(double*));
67      // let each row pointer point to the correct address
68      for(int row=0;row<nrows;row++){
69          array[row] = linear_array + row*ncols;
70      }
71      return array;
72  }
73
74  void
75  destroy_2D_array(
76           double **array
77          )
78  {
79      // free the linear_array
80      free(array[0]);
81      // free the pointers array
82      free(array);
83  }
84
85  void
86  matrix_multiplication(
87                double **result,
88                double **m1,
89                double **m2,
90                unsigned int m,
91                unsigned int n
92               )
93  {
94      // https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication#Schoolbook_algorithm
95      for(int i=0;i<n;i++){
96          for(int j=0;j<n;j++){
97              // calculate matrix element in row i, col j
98              result[i][j] = 0;
99              for(int k=0;k<m;k++){
100                 result[i][j] += m1[i][k] * m2[k][j];
101             }
102         }
103     }
104 }
105
106 double
107 vector_norm(
108         double *v1,
109         unsigned int len
110        )
111 {
112     return sqrt(dot_product(v1, v1, len));
113 }
114
115
116 void
117 normalize_vector(
118          double *v1,
119          unsigned int len
120         )
121 {
122     double norm = vector_norm(v1, len);
123     constant_multiplication(v1, v1, 1./norm, len);
124 }
125
126 double
127 average(
128     double *v1,
129     unsigned int len
```

```
      )
{
    double res = 0;
    for(int i=0;i<len;i++){
        res += v1[i];
    }
    return res/len;
}


double
standard_deviation(
               double *v1,
               unsigned int len
              )
{
    /* https://numpy.org/doc/stable/reference/generated/numpy.std.html
     * The standard deviation is the square root of the average of the squared deviations from the mean,
     * i.e., std = sqrt(mean(x)), where x = abs(a - a.mean())**2.
     * std(v1) = sqrt(sum(v1 - v1_mean)^2 / len(v1)) */
    double mean = average(v1, len);
    double res = 0;
    for(int i=0;i<len;i++){
        res += (v1[i] - mean)*(v1[i] - mean);
    }
    return sqrt(res/len);
}

double
distance_between_vectors(
             double *v1,
             double *v2,
             unsigned int len
                    )
{
    // dist(v1,v2) = |v1 - v2|
    double res = 0;
    for(int i=0;i<len;i++){
        res += (v1[i] - v2[i])*(v1[i] - v2[i]);
    }
    return sqrt(res);
}



void print_vector(double* vec, int length){
    printf("[");
    for(int i=0; i < length; i++){
        printf("%.2f, ", vec[i]);
    }
    printf("\b\b]\n");
}

void fprint_vector(FILE* file, double* vec, int length){
    for(int i=0; i < length; i++){
        fprintf(file, "%.6f", vec[i]);
        if(i<length-1) fprintf(file, ", ");
    }
    fprintf(file, "\n");
}

void print_matrix(double** mat, int n, int m){
    printf("[");
    for(int i=0; i < n; i++){
        printf("[");
        for(int j=0; j < m; j++){
            printf("%.2f, ", mat[i][j]);
        }
        printf("\b\b],");
        if(i<n-1) printf("\n");
    }
    printf("\b]\n");
}
void print_matrix_stack(int n, int m, double mat[][m]){
    printf("[");
    for(int i=0; i < n; i++){
        printf("[");
        for(int j=0; j < m; j++){
            printf("%.2f, ", mat[i][j]);
        }
        printf("\b\b],");
        if(i<n-1) printf("\n");
    }
    printf("\b]\n");
}

void fprint_matrix(FILE* file, double** mat, int n, int m){
    // write matrix to a file
    for(int i=0; i < n; i++){
        for(int j=0; j < m; j++){
            fprintf(file, "%.6f", mat[i][j]);
```

```
            if(j<m-1) fprintf(file, ", ");
            else fprintf(file, "\n");
        }
    }
}


void init_matrix_stack(int n, int m, double mat[n][m], double value){
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            mat[i][j] = value;
        }
    }
}
```

A.3 Plotting of task 1: task1.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def E_pot_fit(a0, c1, c2, c3):
    return c1*a0**2 + c2*a0 + c3

a0, E_pot = np.genfromtxt("data/H1_1.csv", delimiter=",", unpack=True)


# find index of nearest value to the cut point
cut_idx = len(a0)-1 #(np.abs(a0 - 4.03425425)).argmin()

params, pcov = curve_fit(E_pot_fit, a0[:cut_idx], E_pot[:cut_idx])

E_pot_fitted = E_pot_fit(a0, *params)

# minimum point:
a0_min = -0.5*(params[1]/params[0])

plt.figure(figsize=(5,4))
plt.axvline(a0_min, linestyle="--", color="k", alpha=0.5, label=rf"$\mathrm{{argmin}}_{{a_0}}(E_\mathrm{{pot}}) =↩
    {a0_min:.5f} \: \mathrm{{Å}}$")
plt.plot(a0, E_pot_fitted, "--", label="fit")
plt.plot(a0, E_pot, "-", label="simulation data")
plt.xlabel(r"$a_0 \: [\mathrm{Å}]$")
plt.ylabel(r"$E_\mathrm{pot} \: [\mathrm{eV}]$")
plt.legend()
plt.tight_layout()
plt.savefig("plots/H1_1.pdf")
```

A.4 Plotting of task 2: task2.py

```
import numpy as np
import matplotlib.pyplot as plt
import json

def plot_energies_task2(input_file, output_file):
    # get header metadata
    with open(input_file, "r") as file:
        metadata_str = "".join([file.readline() for i in range(4)])
        metadata_str = metadata_str.replace("# ","")
        metadata = json.loads(metadata_str)

    dt = metadata["dt"]

    t, E_pot, E_kin, T, P, a0 = np.genfromtxt(input_file, delimiter=",", unpack=True)

    T_mean = np.mean(T)

    fig, axs = plt.subplots(1,3,figsize=(12,3))
    fig.suptitle(f"T = {T_mean:.5f} K ; dt = {dt*1e3:.1f} fs")

    plt.sca(axs[0])
    plt.plot(t, E_pot, "C0", label=r"$E_\mathrm{pot}$")
    plt.xlabel("time [ps]")
    plt.ylabel("potential energy [eV]")
    plt.legend()

    plt.sca(axs[1])
    plt.plot(t, E_kin, "C1", label=r"$E_\mathrm{kin}$")
    plt.xlabel("time [ps]")
    plt.ylabel("kinetic energy [eV]")
    plt.legend()
```

```
    plt.sca(axs[2])
    plt.plot(t, E_pot+E_kin, "C2", label=r"$E_\mathrm{tot}$")
    plt.xlabel("time [ps]")
    plt.ylabel("total energy [eV]")
    plt.legend()


    plt.tight_layout()
    plt.savefig(output_file)

plot_energies_task2("data/H1_2_small_enough.csv", "plots/H1_2_small_enough.pdf")
plot_energies_task2("data/H1_2_far_too_large.csv", "plots/H1_2_far_too_large.pdf")
plot_energies_task2("data/H1_2_little_too_large.csv", "plots/H1_2_little_too_large.pdf")
```

A.5 Plotting of task 3: task3.py

```
import numpy as np
import matplotlib.pyplot as plt
import json
from task2 import plot_energies_task2


t, E_pot, E_kin, T, P, a0 = np.genfromtxt("data/H1_3_temp_scaling.csv", delimiter=",", unpack=True)
t2, E_pot2, E_kin2, T2, P2, a02 = np.genfromtxt("data/H1_3_pressure_scaling.csv", delimiter=",", unpack=True)

E_tot = E_kin + E_pot
E_tot2 = E_kin2 + E_pot2

# get header metadata
with open("data/H1_3_pressure_scaling.csv", "r") as file:
    metadata_str = "".join([file.readline() for i in range(4)])
    metadata_str = metadata_str.replace("# ","")
    metadata = json.loads(metadata_str)

T_desired = metadata["T_desired"]
P_desired = metadata["P_desired"]

t_switch_on_P_scaling = (t[-1]+t2[0])/2

fig, axs = plt.subplots(4,1,figsize=(8,8))

# plot the temperature
plt.sca(axs[0])
plt.axvline(t_switch_on_P_scaling, linestyle="--", color="k", alpha=0.5)#, label=f"start pressure \nequilibration↩
    \nat $t={t_switch_on_P_scaling:.2f} \\:\mathrm{{ps}}$")
plt.plot(t, T, "C0")
plt.plot(t2, T2, "C0")
plt.axhline(T_desired, linestyle=":", color="k", label=f"desired $T={T_desired:.2f} \\: \\mathrm{{K}}$")
plt.xlabel("time [ps]")
plt.ylabel("temperature [K]")
plt.legend(loc="upper right")

# plot the pressure
plt.sca(axs[1])
plt.axvline(t_switch_on_P_scaling, linestyle="--", color="k", alpha=0.5)#, label=f"start pressure \nequilibration↩
    \nat $t={t_switch_on_P_scaling:.2f} \\:\mathrm{{ps}}$")
plt.plot(t, P, "C1")
plt.plot(t2, P2, "C1")
plt.axhline(P_desired, linestyle=":", color="k", label=f"desired $P={P_desired:.2f} \\: \\mathrm{{bar}}$")
plt.xlabel("time [ps]")
plt.ylabel("pressure [bar]")
plt.legend(loc="upper right")

# plot the total energy
plt.sca(axs[2])
plt.axvline(t_switch_on_P_scaling, linestyle="--", color="k", alpha=0.5)#, label=f"start pressure \nequilibration↩
    \nat $t={t_switch_on_P_scaling:.2f} \\:\mathrm{{ps}}$")
plt.plot(t, E_tot, "C2")
plt.plot(t2, E_tot2, "C2")
plt.axhline(E_tot2[-1], linestyle=":", color="k", label=f"end $E_\mathrm{{tot}}={E_tot2[-1]:.2f} \\: \\mathrm{{eV↩
    }}$")
plt.xlabel("time [ps]")
plt.ylabel("total energy [eV]")
plt.legend(loc="lower right")

# plot the lattice constant
plt.sca(axs[3])
plt.axvline(t_switch_on_P_scaling, linestyle="--", color="k", alpha=0.5)#, label=f"start pressure \nequilibration↩
    \nat $t={t_switch_on_P_scaling:.2f} \\:\mathrm{{ps}}$")
plt.plot(t, a0, "C3")
plt.plot(t2, a02, "C3")
plt.axhline(a02[-1], linestyle=":", color="k", label=f"end $a_0={a02[-1]:.4f} \\: \\mathrm{{Å}}$")
plt.xlabel("time [ps]")
plt.ylabel("lattice constant [Å]")
plt.legend(loc="lower right")
```

```
65
66  plt.tight_layout()
67  plt.savefig("plots/H1_3_equilibration.pdf")
68
69
70
71  ############################################################
72  # Plot the simulation after the equilibration
73
74
75  # get header metadata
76  with open("data/H1_3_after_scaling.csv", "r") as file:
77      metadata_str = "".join([file.readline() for i in range(4)])
78      metadata_str = metadata_str.replace("# ","")
79      metadata = json.loads(metadata_str)
80
81  data = np.genfromtxt("data/H1_3_after_scaling.csv", delimiter=",",unpack=True)
82
83  t, E_pot, E_kin, T, P, a0 = data[:6]
84  E_tot = E_kin + E_pot
85  pos = data[6:]
86  x = pos[::3]
87  y = pos[1::3]
88  z = pos[2::3]
89
90  # plot the positions
91  fig, axs = plt.subplots(3,1, figsize=(8,6))
92
93  plt.sca(axs[0])
94  for i,x_i in enumerate(x):
95      plt.plot(t, x_i[0]-x_i, label=f"x_{i}")
96
97  plt.xlabel("time [ps]")
98  plt.ylabel("$x(t=0) - x(t)$ [Å]")
99  plt.legend(loc="upper right")
100
101 plt.sca(axs[1])
102 for i,y_i in enumerate(y):
103     plt.plot(t, y_i[0]-y_i, label=f"y_{i}")
104
105 plt.xlabel("time [ps]")
106 plt.ylabel("$y(t=0) - y(t)$ [Å]")
107 plt.legend(loc="upper right")
108
109 plt.sca(axs[2])
110 for i,z_i in enumerate(z):
111     plt.plot(t, z_i[0]-z_i, label=f"z_{i}")
112
113 plt.xlabel("time [ps]")
114 plt.ylabel("$z(t=0) - z(t)$ [Å]")
115 plt.legend(loc="upper right")
116
117 plt.tight_layout()
118 plt.savefig("plots/H1_3_positions.pdf")
119
120
121
122 ############################################################
123 # plot the equilibration plots for the simulation without scaling
124 fig, axs = plt.subplots(4,1,figsize=(8,8))
125
126 # plot the temperature
127 plt.sca(axs[0])
128 plt.plot(t, T, "C0")
129 plt.axhline(T_desired, linestyle=":", alpha=0.5, color="k", label=f"desired $T={T_desired:.2f} \\: \\mathrm{{K}}$")
130 plt.axhline(np.mean(T), linestyle="--", alpha=0.5, color="k", label=f"mean $T={np.mean(T):.2f} \\: \\mathrm{{K}}$")
131 plt.plot(t[0],T[0],alpha=0., label=f"std $T={np.std(T):.2f} \\: \\mathrm{{K}}$")
132 plt.xlabel("time [ps]")
133 plt.ylabel("temperature [K]")
134 plt.legend(loc="upper right")
135
136 # plot the pressure
137 plt.sca(axs[1])
138 plt.plot(t, P, "C1")
139 plt.axhline(P_desired, linestyle=":", alpha=0.5, color="k", label=f"desired $P={P_desired:.2f} \\: \\mathrm{{bar}}$")
140 plt.axhline(np.mean(P), linestyle="--", alpha=0.5, color="k", label=f"mean $P={np.mean(P):.2f} \\: \\mathrm{{bar}}$")
141 plt.plot(t[0],P[0],alpha=0., label=f"std $P={np.std(P):.2f} \\: \\mathrm{{bar}}$")
142 plt.xlabel("time [ps]")
143 plt.ylabel("pressure [bar]")
144 plt.legend(loc="upper right")
145
146 # plot the total energy
147 plt.sca(axs[2])
148 plt.plot(t, E_tot, "C2")
149 plt.axhline(E_tot2[-1], linestyle=":", alpha=0.5, color="k", label=f"end $E_\mathrm{{tot}}={E_tot2[-1]:.2f} \\: \\mathrm{{eV}}$")
```

```
plt.axhline(np.mean(E_tot), linestyle="--", alpha=0.5, color="k", label=f"mean $E_\mathrm{{tot}}={np.mean(E_tot):.2f} \\: \\mathrm{{eV}}$")
plt.plot(t[0],E_tot[0],alpha=0., label=f"std $E_\mathrm{{tot}}={np.std(E_tot):.4f} \\: \\mathrm{{eV}}$")
plt.xlabel("time [ps]")
plt.ylabel("total energy [eV]")
plt.legend(loc="lower right")

# plot the lattice constant
plt.sca(axs[3])
plt.plot(t, a0, "C3")
plt.axhline(a02[-1], linestyle=":", alpha=0.5, color="k", label=f"end $a_0={a02[-1]:.4f} \\: \\mathrm{{Å}}$")
plt.axhline(np.mean(a0), linestyle="--", alpha=0.5, color="k", label=f"mean $a_0={np.mean(a0):.4f} \\: \\mathrm{{Å}}$")
plt.xlabel("time [ps]")
plt.ylabel("lattice constant [Å]")
plt.legend(loc="lower right")

plt.tight_layout()
plt.savefig("plots/H1_3_after_equilibration.pdf")
```

A.6 Plotting of task 4: task4.py

```
import numpy as np
import matplotlib.pyplot as plt
import json
from task2 import plot_energies_task2


t, E_pot, E_kin, T, P, a0 = np.genfromtxt("data/H1_4_temp_scaling.csv", delimiter=",", unpack=True)
t2, E_pot2, E_kin2, T2, P2, a02 = np.genfromtxt("data/H1_4_pressure_scaling.csv", delimiter=",", unpack=True)
t3, E_pot3, E_kin3, T3, P3, a03 = np.genfromtxt("data/H1_4_temp_decreasing.csv", delimiter=",", unpack=True)

E_tot = E_kin + E_pot
E_tot2 = E_kin2 + E_pot2
E_tot3 = E_kin3 + E_pot3

# get header metadata
with open("data/H1_4_temp_scaling.csv", "r") as file:
    metadata_str = "".join([file.readline() for i in range(4)])
    metadata_str = metadata_str.replace("# ","")
    metadata = json.loads(metadata_str)

T_melting = metadata["T_desired"]

# get header metadata
with open("data/H1_4_temp_decreasing.csv", "r") as file:
    metadata_str = "".join([file.readline() for i in range(4)])
    metadata_str = metadata_str.replace("# ","")
    metadata = json.loads(metadata_str)

T_desired = metadata["T_desired"]
P_desired = metadata["P_desired"]

t_switch_on_P_scaling = (t[-1]+t2[0])/2
t_switch_on_T_decreasing = (t2[-1]+t3[0])/2

fig, axs = plt.subplots(4,1,figsize=(8,8))

# plot the temperature
plt.sca(axs[0])
plt.axvline(t_switch_on_P_scaling, linestyle="--", color="k", alpha=0.5)#, label=f"start pressure \nequilibration\nat $t={t_switch_on_P_scaling:.2f} \\:\mathrm{{ps}}$")
plt.axvline(t_switch_on_T_decreasing, linestyle="--", color="k", alpha=0.5)
plt.plot(t, T, "C0")
plt.plot(t2, T2, "C0")
plt.plot(t3, T3, "C0")
plt.axhline(T_desired, linestyle=":", color="k", label=f"desired $T={T_desired:.2f} \\: \\mathrm{{K}}$")
plt.axhline(T_melting, linestyle=":", color="k", label=f"melting $T={T_melting:.2f} \\: \\mathrm{{K}}$")
plt.xlabel("time [ps]")
plt.ylabel("temperature [K]")
plt.legend(loc="upper right")

# plot the pressure
plt.sca(axs[1])
plt.axvline(t_switch_on_P_scaling, linestyle="--", color="k", alpha=0.5)#, label=f"start pressure \nequilibration\nat $t={t_switch_on_P_scaling:.2f} \\:\mathrm{{ps}}$")
plt.axvline(t_switch_on_T_decreasing, linestyle="--", color="k", alpha=0.5)
plt.plot(t, P, "C1")
plt.plot(t2, P2, "C1")
plt.plot(t3, P3, "C1")
plt.axhline(P_desired, linestyle=":", color="k", label=f"desired $P={P_desired:.2f} \\: \\mathrm{{bar}}$")
plt.xlabel("time [ps]")
plt.ylabel("pressure [bar]")
plt.legend(loc="upper right")

# plot the total energy
```

```
plt.sca(axs[2])
plt.axvline(t_switch_on_P_scaling, linestyle="--", color="k", alpha=0.5)#, label=f"start pressure \nequilibration\nat $t={t_switch_on_P_scaling:.2f} \\:\mathrm{{ps}}$")
plt.axvline(t_switch_on_T_decreasing, linestyle="--", color="k", alpha=0.5)
plt.plot(t, E_tot, "C2")
plt.plot(t2, E_tot2, "C2")
plt.plot(t3, E_tot3, "C2")
plt.axhline(E_tot3[-1], linestyle=":", color="k", label=f"end $E_\mathrm{{tot}}={E_tot3[-1]:.2f} \\: \\mathrm{{eV}}$")
plt.xlabel("time [ps]")
plt.ylabel("total energy [eV]")
plt.legend(loc="lower right")

# plot the lattice constant
plt.sca(axs[3])
plt.axvline(t_switch_on_P_scaling, linestyle="--", color="k", alpha=0.5)#, label=f"start pressure \nequilibration\nat $t={t_switch_on_P_scaling:.2f} \\:\mathrm{{ps}}$")
plt.axvline(t_switch_on_T_decreasing, linestyle="--", color="k", alpha=0.5)
plt.plot(t, a0, "C3")
plt.plot(t2, a02, "C3")
plt.plot(t3, a03, "C3")
plt.axhline(a03[-1], linestyle=":", color="k", label=f"end $a_0={a03[-1]:.4f} \\: \\mathrm{{Å}}$")
plt.xlabel("time [ps]")
plt.ylabel("lattice constant [Å]")
plt.legend(loc="lower right")

plt.tight_layout()
plt.savefig("plots/H1_4_equilibration.pdf")



############################################################
# Plot the simulation after the equilibration


# get header metadata
with open("data/H1_4_after_scaling.csv", "r") as file:
    metadata_str = "".join([file.readline() for i in range(4)])
    metadata_str = metadata_str.replace("# ","")
    metadata = json.loads(metadata_str)

data = np.genfromtxt("data/H1_4_after_scaling.csv", delimiter=",",unpack=True)

t, E_pot, E_kin, T, P, a0 = data[:6]
E_tot = E_kin + E_pot
pos = data[6:]
x = pos[::3]
y = pos[1::3]
z = pos[2::3]

# plot the positions
fig, axs = plt.subplots(3,1, figsize=(8,6))

plt.sca(axs[0])
for i,x_i in enumerate(x):
    plt.plot(t, x_i[0]-x_i, label=f"x_{i}")

plt.xlabel("time [ps]")
plt.ylabel("$x(t=0) - x(t)$ [Å]")
plt.legend(loc="upper right")

plt.sca(axs[1])
for i,y_i in enumerate(y):
    plt.plot(t, y_i[0]-y_i, label=f"y_{i}")

plt.xlabel("time [ps]")
plt.ylabel("$y(t=0) - y(t)$ [Å]")
plt.legend(loc="upper right")

plt.sca(axs[2])
for i,z_i in enumerate(z):
    plt.plot(t, z_i[0]-z_i, label=f"z_{i}")

plt.xlabel("time [ps]")
plt.ylabel("$z(t=0) - z(t)$ [Å]")
plt.legend(loc="upper right")

plt.tight_layout()
plt.savefig("plots/H1_4_positions.pdf")



############################################################
# plot the equilibration plots for the simulation without scaling
fig, axs = plt.subplots(4,1,figsize=(8,8))

# plot the temperature
plt.sca(axs[0])
plt.plot(t, T, "C0")
```

```
plt.axhline(T_desired, linestyle=":", alpha=0.5, color="k", label=f"desired $T={T_desired:.2f} \\: \\mathrm{{K}}$")
plt.axhline(np.mean(T), linestyle="--", alpha=0.5, color="k", label=f"mean $T={np.mean(T):.2f} \\: \\mathrm{{K}}$")
plt.plot(t[0],T[0],alpha=0., label=f"std $T={np.std(T):.2f} \\: \\mathrm{{K}}$")
plt.xlabel("time [ps]")
plt.ylabel("temperature [K]")
plt.legend(loc="upper right")

# plot the pressure
plt.sca(axs[1])
plt.plot(t, P, "C1")
plt.axhline(P_desired, linestyle=":", alpha=0.5, color="k", label=f"desired $P={P_desired:.2f} \\: \\mathrm{{bar}}$")
plt.axhline(np.mean(P), linestyle="--", alpha=0.5, color="k", label=f"mean $P={np.mean(P):.2f} \\: \\mathrm{{bar}}$")
plt.plot(t[0],P[0],alpha=0., label=f"std $P={np.std(P):.2f} \\: \\mathrm{{bar}}$")
plt.xlabel("time [ps]")
plt.ylabel("pressure [bar]")
plt.legend(loc="upper right")

# plot the total energy
plt.sca(axs[2])
plt.plot(t, E_tot, "C2")
plt.axhline(E_tot3[-1], linestyle=":", alpha=0.5, color="k", label=f"end $E_\mathrm{{tot}}={E_tot3[-1]:.2f} \\: \\mathrm{{eV}}$")
plt.axhline(np.mean(E_tot), linestyle="--", alpha=0.5, color="k", label=f"mean $E_\mathrm{{tot}}={np.mean(E_tot):.2f}  \\: \\mathrm{{eV}}$")
plt.plot(t[0],E_tot[0],alpha=0., label=f"std $E_\mathrm{{tot}}={np.std(E_tot):.4f} \\: \\mathrm{{eV}}$")
plt.xlabel("time [ps]")
plt.ylabel("total energy [eV]")
plt.legend(loc="lower right")

# plot the lattice constant
plt.sca(axs[3])
plt.plot(t, a0, "C3")
plt.axhline(a03[-1], linestyle=":", alpha=0.5, color="k", label=f"end $a_0={a03[-1]:.4f} \\: \\mathrm{{Å}}$")
plt.axhline(np.mean(a0), linestyle="--", alpha=0.5, color="k", label=f"mean $a_0={np.mean(a0):.4f} \\: \\mathrm{{Å}}$")
plt.xlabel("time [ps]")
plt.ylabel("lattice constant [Å]")
plt.legend(loc="lower right")

plt.tight_layout()
plt.savefig("plots/H1_4_after_equilibration.pdf")
```

A.7 Plotting of task 5: task5.py

```
import numpy as np
import matplotlib.pyplot as plt
import json

eV = 1.602176634e-19 # J
celsius = 273.15 # K
k_B = 8.617333262e-5 # eV/K
m_u = 1.660539066e-27 # kg

m_Al = 26.98153853 * m_u * 10**3 # g

def C_V(E,N,T):
    E_var = np.var(E)
    return 3*N*k_B/2 / (1 - 2*E_var/(3*N*k_B**2*T**2))

def C_V_specific(E,N,T):
    C_V_ = C_V(E,N,T) # eV/K
    return C_V_ * eV / (N*m_Al) # J/(g K)


# solid state
data = np.genfromtxt("data/H1_3_after_scaling.csv", delimiter=",",unpack=True)

with open("data/H1_3_pressure_scaling.csv", "r") as file:
    metadata_str = "".join([file.readline() for i in range(4)])
    metadata_str = metadata_str.replace("# ","")
    metadata = json.loads(metadata_str)

N = metadata["n_atoms"]
T_desired = metadata["T_desired"]

t, E_pot, E_kin, T, P, a0 = data[:6]
E_tot = E_kin + E_pot
pos = data[6:]
x = pos[::3]
y = pos[1::3]
z = pos[2::3]

```

```
T_mean = np.mean(T)
T_std = np.std(T)

print("Task 5 - Solid state:")
print(f"T_desired = {T_desired:.2f} K ; T_mean = {T_mean:.2f} +- {T_std:.2f} K")
print(f"T_desired = {T_desired-celsius:.2f} °C ; T_mean = {T_mean-celsius:.2f} +- {T_std:.2f} °C")
print(f"C_V (E_kin, T_mean)     = {C_V(E_kin,N,T_mean):.6f} eV/K")
print(f"C_V (E_pot, T_mean)     = {C_V(E_pot,N,T_mean):.6f} eV/K")
print(f"c_V (E_kin, T_mean)     = {C_V_specific(E_kin,N,T_mean):.6f} J/(g K)")
print(f"c_V (E_pot, T_mean)     = {C_V_specific(E_pot,N,T_mean):.6f} J/(g K)")
print(f"C_V (E_kin, T_desired) = {C_V(E_kin,N,T_desired):.6f} eV/K")
print(f"C_V (E_pot, T_desired) = {C_V(E_pot,N,T_desired):.6f} eV/K")
print(f"c_V (E_kin, T_desired) = {C_V_specific(E_kin,N,T_desired):.6f} J/(g K)")
print(f"c_V (E_pot, T_desired) = {C_V_specific(E_pot,N,T_desired):.6f} J/(g K)")


# fluid state
data = np.genfromtxt("data/H1_4_after_scaling.csv", delimiter=",",unpack=True)

with open("data/H1_4_temp_decreasing.csv", "r") as file:
    metadata_str = "".join([file.readline() for i in range(4)])
    metadata_str = metadata_str.replace("# ","")
    metadata = json.loads(metadata_str)

N = metadata["n_atoms"]
T_desired = metadata["T_desired"]

t, E_pot, E_kin, T, P, a0 = data[:6]
E_tot = E_kin + E_pot
pos = data[6:]
x = pos[::3]
y = pos[1::3]
z = pos[2::3]

T_mean = np.mean(T)
T_std = np.std(T)

print("Task 5 - Liquid state:")
print(f"T_desired = {T_desired:.2f} K ; T_mean = {T_mean:.2f} +- {T_std:.2f} K")
print(f"T_desired = {T_desired-celsius:.2f} °C ; T_mean = {T_mean-celsius:.2f} +- {T_std:.2f} °C")
print(f"C_V (E_kin, T_mean)     = {C_V(E_kin,N,T_mean):.6f} eV/K")
print(f"C_V (E_pot, T_mean)     = {C_V(E_pot,N,T_mean):.6f} eV/K")
print(f"c_V (E_kin, T_mean)     = {C_V_specific(E_kin,N,T_mean):.6f} J/(g K)")
print(f"c_V (E_pot, T_mean)     = {C_V_specific(E_pot,N,T_mean):.6f} J/(g K)")
print(f"C_V (E_kin, T_desired) = {C_V(E_kin,N,T_desired):.6f} eV/K")
print(f"C_V (E_pot, T_desired) = {C_V(E_pot,N,T_desired):.6f} eV/K")
print(f"c_V (E_kin, T_desired) = {C_V_specific(E_kin,N,T_desired):.6f} J/(g K)")
print(f"c_V (E_pot, T_desired) = {C_V_specific(E_pot,N,T_desired):.6f} J/(g K)")
```

A.8 Plotting of task 6: task6.py

```
import numpy as np
import matplotlib.pyplot as plt
import json

# get header metadata
with open("data/H1_6.csv", "r") as file:
    metadata_str = "".join([file.readline() for i in range(1)])
    metadata_str = metadata_str.replace("# ","")
    metadata = json.loads(metadata_str)

n_atoms = metadata["n_atoms"]
L_box = metadata["L_box[Å]"]
V = L_box**3

r, N_r = np.genfromtxt("data/H1_6.csv", delimiter=",", unpack=True)

n_bins = len(r)
dr = np.diff(r)[0]
r_max = r[-1] + dr/2

print(f"n_bins = {n_bins} ; r_max = {r_max:.4f} Å ; dr = {dr:.4f} Å")

N_ideal = [(n_atoms-1)*4*np.pi*(3*k**2-3*k+1)*dr**3/(3*V) for k in range(1,n_bins+1)]
N_ideal = np.array(N_ideal)

g_r = N_r/N_ideal

# find the first local minimum
r_m_idx = np.argmin(g_r[np.argmax(g_r):])+np.argmax(g_r)
r_m = r[r_m_idx]

I_r_m = 4*np.pi*(n_atoms/V)*np.trapz(g_r[:r_m_idx]*r[:r_m_idx]**2 ,r[:r_m_idx])

plt.figure(figsize=(5,4))
plt.axvline(r_m, linestyle="--", color="k", alpha=0.5, label=f"$r_m = {r_m:.4f} \\: \\mathrm{{Å}}$")
```

```
plt.fill_between(r[:r_m_idx],0,g_r[:r_m_idx], color="C1", alpha=0., label=f"$\\mathcal{{I}}(r_m) = {I_r_m:.4f}$")
plt.plot(r,g_r)
plt.xlabel(r"$r \: / \: \mathrm{Å}$")
plt.ylabel(r"$g(r)$")
plt.legend()
plt.tight_layout()
plt.savefig("plots/H1_6.pdf")
```

A.9 Plotting of task 7: task7.py

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.stats import binned_statistic

q_x, q_y, q_z, q, S_q = np.genfromtxt("data/H1_7.csv", delimiter=",", unpack=True)

df = pd.DataFrame(np.column_stack([q,S_q]), columns=["q","S_q"])
temp = df.groupby('q').mean().reset_index().values
q_mean = temp[:,0]
S_q_mean = temp[:,1]
temp = df.groupby('q').std().reset_index().values
S_q_std = temp[:,1]

q_min = 0.1
q_max = np.max(q)
n_bins = 300
bin_edges = np.linspace(q_min, q_max, n_bins+1)
bin_width = np.diff(bin_edges)[0]
bin_centers = bin_edges[:-1] + bin_width/2

S_q_binned_mean = binned_statistic(q,S_q,"mean",bin_edges)[0]
S_q_binned_std = binned_statistic(q,S_q,"std",bin_edges)[0]

plt.figure(figsize=(5,4))
#plt.plot(q[1:], S_q[1:], "x",alpha=0.3, label=r"$S(q)$")
plt.plot(q_mean[1:], S_q_mean[1:], "C1.", markersize=1., label="$<S(q)>$")
#plt.fill_between(q_mean,S_q_mean-S_q_std, S_q_mean+S_q_std, color="C0", alpha=0.2, label="uncertainty")
plt.fill_between(bin_edges[:-1],S_q_binned_mean-S_q_binned_std, S_q_binned_mean+S_q_binned_std, step="post", ↩
    color="C0", alpha=0.2, label="binned std")
plt.step(bin_edges[:-1], S_q_binned_mean, where="post", label="binned mean")
plt.xlabel("$q \: / \: \mathrm{Å^-1}$")
plt.ylabel("$S(q)$")
plt.legend()
plt.tight_layout()
plt.savefig("plots/H1_7.pdf")
```