

# Homework 1:

## Simulation of Molecular Dynamics

Jonatan Haraldsson  
[jonhara@chalmers.se](mailto:jonhara@chalmers.se)

Oscar Stommendal  
[oscarsto@chalmers.se](mailto:oscarsto@chalmers.se)

Task N <sup>o</sup>	Points	Avail. points
$\Sigma$		

December 6<sup>th</sup> 2024  
Course: *Computational Physics*, 7.5 hp  
Course Code: *FKA 122*

*Physics, MSc.*  
CHALMERS UNIVERSITY OF TECHNOLOGY



# Introduction

Already in antiquity people studied the effect of particles impinging on other particles. Since then the art has developed, however, when modeling larger so-called many-body systems, analytical solutions are often complex. Consequently, with the rise of modern computers, several numerical algorithms for solving the equations of motion for such a system have been developed. One early example of which is the FPUT model, an an-harmonic model for studying dynamics in a solid, developed and tested during the 1950s by Fermi, Pasta, Ulam and Tsinguo [1]. In recent years advanced machine learning algorithms using neural networks gives even greater insight into the study of molecular dynamics.

In this report, we present results from simulations done on 256 interacting aluminum atoms in a lattice represented by a  $4 \times 4 \times 4$  supercell (i.e. 64 unit cells) using a Neuroevolution potential (NEP) [2]. Apart from simulating the motion of each atom, system properties such as energy, temperature and pressure was calculated. Furthermore, the heat capacity  $C_V$ , radial distribution function,  $g(r)$ , and structure factor,  $S(q)$ , were also calculated. All simulations were done in `C`, however, plots and some minor calculations were done in `Python`. For reference, the source code is available in Appendix B and on [GitHub](#).

## Task 1 – Potential Energy and Unit Cell Volume

In the first Task, FCC structures with different lattice parameters  $a_0$  were generated with the provided function `init_fcc`. Specifically, a vector with lattice parameters between 4.0–4.08 Å was used and potential energies were then obtained through `get_energy_AL(pos, a0[i] * cell_length, N)`, where `cell_length` = 4 and `N` = 256. Lastly, the energies along with a quadratic fit were plotted, see Figure 1. The minimum energy per unit cell volume was given by  $a_0 = 4.03$  Å. Here, the system is initialized with zero kinetic energy, or, equivalently,  $T = 0$  K. In the following tasks,  $v_{initial} = 0$  for all atoms, which made the found value of  $a_0 = 4.03$  Å at  $T = 0$  K crucial since it was set a initial lattice constant in all simulations. This value is reasonable as the lattice constant for Al at room temperature is known to be 4.05 Å, and a lower temperature should decrease this value to some degree [3]. Moreover, [4] found the lattice parameter of Al to be 4.032 Å at 0 K, which further strengthens this result.

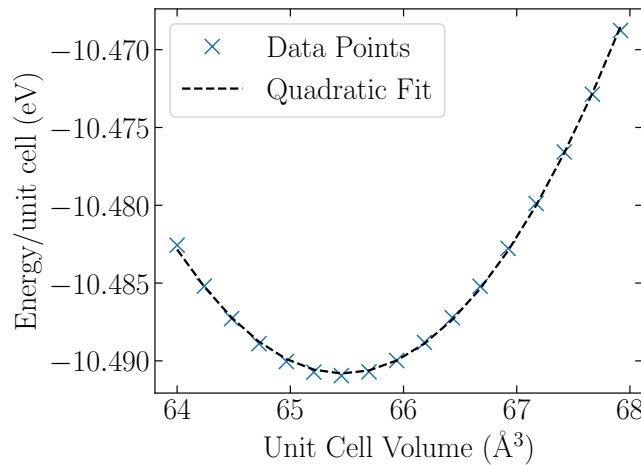


Figure 1: Potential energy per unit cell for initialized FCC structures with different lattice parameter  $a_0$  (note that we have  $V = a_0^3$  on the  $x$ -axis), with a quadratic fit.

## Task 2 – Energy Conservation and Choice of Time-step

In the second task, the equations of motion were solved with slightly disturbed initial conditions. To update positions and velocities for all atoms, the *Verlet algorithm* was realized. In short, the *Verlet algorithm* was originally derived by a symmetric Taylor expansion forwards and backwards in time using a time-step  $\Delta t$ . However, to avoid storing two time instances ( $t \pm \Delta t$ ) in each iteration, a modified version, where the velocity is updated in two half-steps, was used. Forces and virials, obtained from the provided function `calculate`, were re-calculated in between the two velocity update steps, according to the provided code snippet below. Prior to simulation, the initial positions were disturbed with a factor  $\pm 6.5\%$  of the lattice spacing  $a_0 = 4.03 \text{ \AA}$  found in Task 1 using `gsl_rng` (random generator)\*.

### Code Snippet Task 2, Verlet

```
1  // Calculate the initial forces, potential energy and virial
2  calculate(&E_pot, &virial, forces, positions, a_0 * cell_length, N);
3  for (unsigned int i = 0; i < its; i++)
4  {
5      E_kin = 0.0;
6      // Perform the first half step
7      for (unsigned int j = 0; j < N; j++)
8      {
9          for (unsigned int k = 0; k < 3; k++)
10         {
11             velocities[j][k] += 0.5 * delta_t * forces[j][k] / m;
12             positions[j][k] += delta_t * velocities[j][k];
13         }
14     }
15     // Calculate new accelerations
16     calculate(&E_pot, &virial, forces, positions, a_0 * cell_length, N);
17     // Perform the second half step
18     for (unsigned int j = 0; j < N; j++)
19     {
20         for (unsigned int k = 0; k < 3; k++)
21         {
22             velocities[j][k] += 0.5 * delta_t * forces[j][k] / m;
23         }
24         E_kin += 0.5 * m * vector_norm(velocities[j], 3) *
25             ↪ vector_norm(velocities[j], 3);
26     }
```

To find a fitting time-step ( $\Delta t$ ) for the simulation, the system's energy, i.e. potential and kinetic, and temperature were plotted for different  $\Delta t$ . This is done for  $\Delta t = 0.1 \text{ ps}$  in Figure 2,  $\Delta t = 0.02 \text{ ps}$  in Figure 3 and  $\Delta t = 0.001 \text{ ps}$  in Figure 4. Comparing the Figures,  $\Delta t = 0.001 \text{ ps}$  seems to be sufficiently small to conserve the energy. To the right in Figures 2, 3 and 4, the average temperature during the simulation is plotted, and  $\Delta t = 0.001 \text{ ps}$  gives, again, a stable result at  $\sim 620^\circ\text{C}$  ( $900 \text{ K}$ ). For the slightly larger  $\Delta t = 0.02 \text{ ps}$ , the average temperature is diverging after  $\sim 2.5 \text{ ps}$ , while  $\Delta t = 0.1 \text{ ps}$  diverges instantly. This is probably reasonable since a time-step too large cannot capture the atomic oscillations, leading to divergence. For the moderately high time-step  $\Delta t = 0.02 \text{ ps}$  the algorithm seems stable at first, however, errors are probably accumulating up to the point where we see divergence.

---

\*For reference, the same random generating seed was used for all simulations.

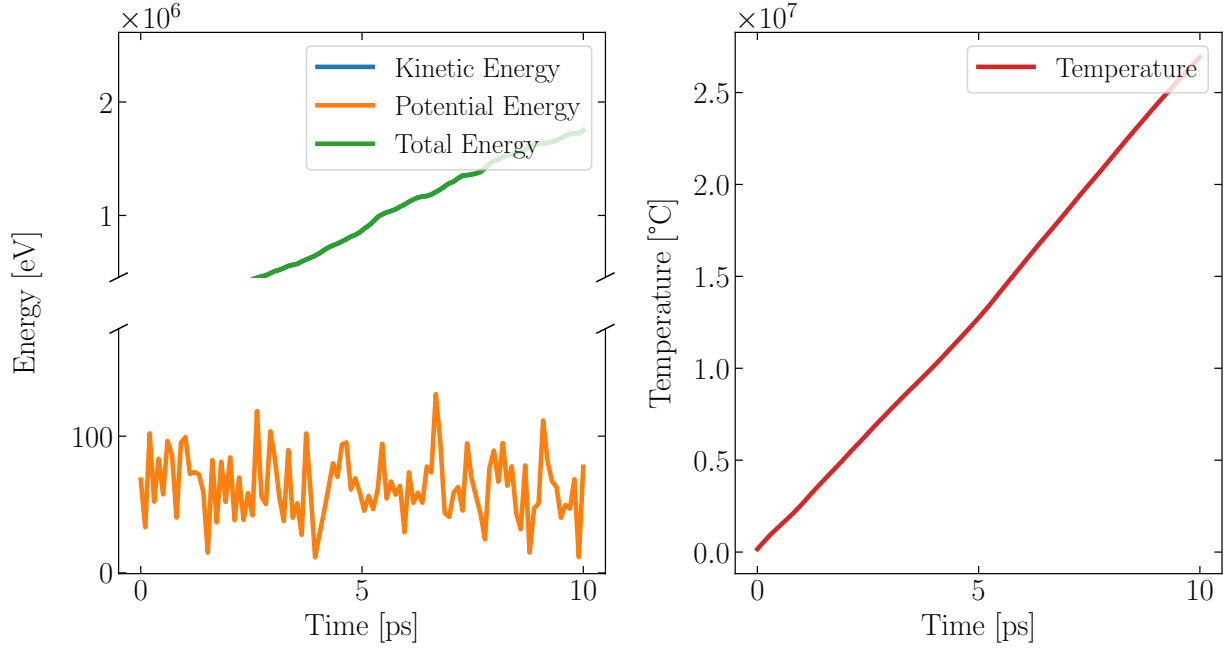


Figure 2: The plot to the left contain instantaneous kinetic, potential and total energy for the system at each time-step. The right plot displays the average temperature throughout the simulation. In this case,  $\Delta t = 0.1$  ps.

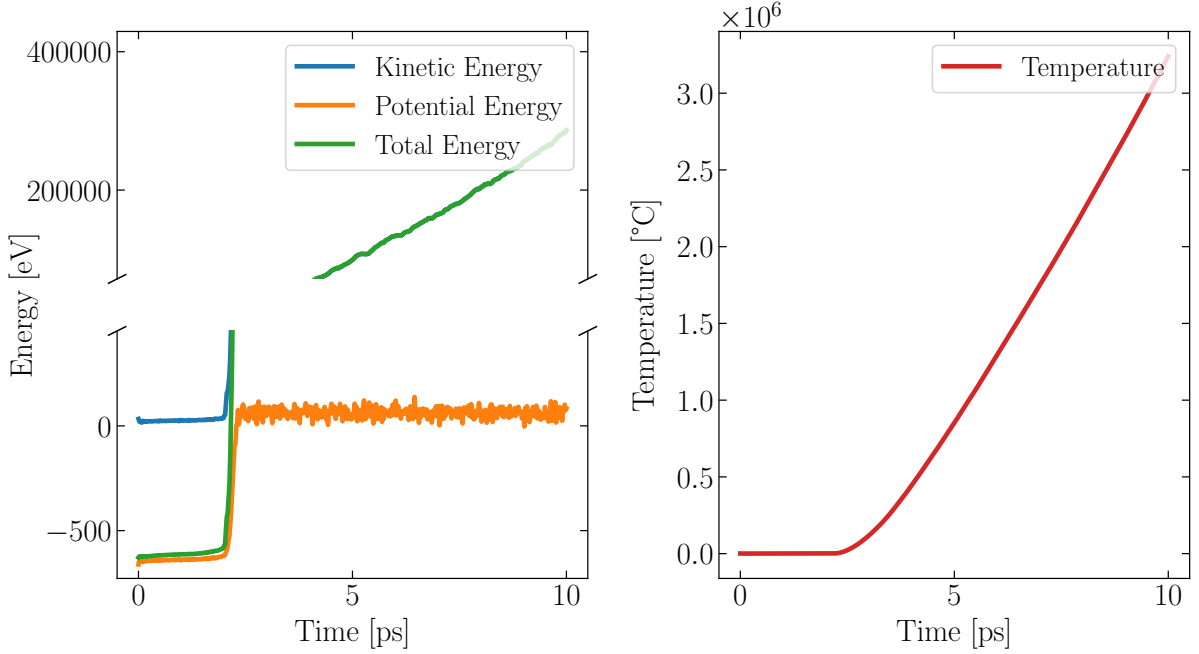


Figure 3: The plot to the left contain instantaneous kinetic, potential and total energy for the system at each time-step. The right plot displays the average temperature throughout the simulation. In this case,  $\Delta t = 0.02$  ps.

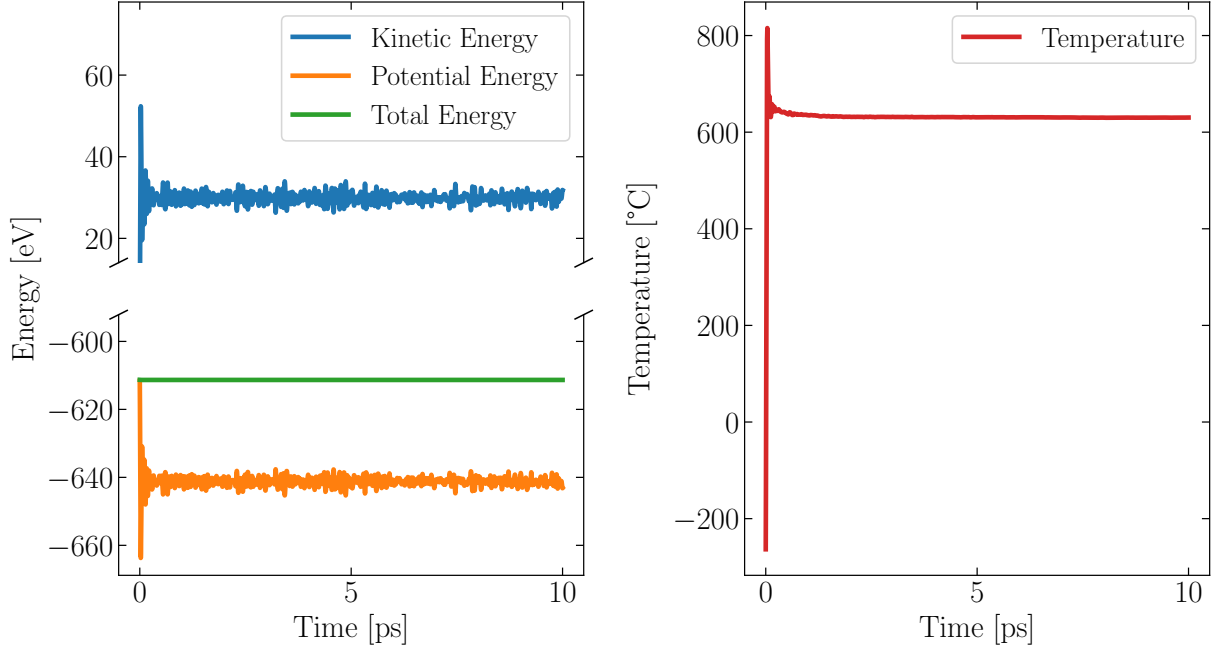


Figure 4: The plot to the left contain instantaneous kinetic, potential and total energy for the system at each time-step. The right plot displays the average temperature throughout the simulation. In this case,  $\Delta t = 0.001$  ps.

### Task 3 – Characteristics of the Solid System

For this task, code to equilibrate the system at a particular temperature and pressure prior to the production was implemented. With this equilibrium set-up, temperature and pressure should roughly converge exponentially to specified  $T_{eq}$  or  $P_{eq}$ . The convergence rate is governed by the time constants  $\tau_T$  or  $\tau_P$ . For both  $T$  and  $P$ , the relation  $\mathcal{A}(t) = A_{eq} + (\mathcal{A}(0) - A_{eq})e^{-t/\tau_A}$  was used, where  $\mathcal{A}(t)$  is the instantaneous value of the dummy variable  $A$ . Since  $T$  is related to average velocities, and  $P$  to positions, the velocity  $\mathbf{v}$  and position  $\mathbf{r}$  for each particle was scaled accordingly

$$\mathbf{v}_i^{new} = \alpha_T^{1/2} \mathbf{v}_i^{old} \quad \text{and} \quad \mathbf{r}_i^{new} = \alpha_P^{1/3} \mathbf{r}_i^{new}$$

at the end of each iteration  $i$  in the *Verlet* function. In addition, the scaling parameters  $\alpha_T$  and  $\alpha_P$  are given by

$$\alpha_T(t) = 1 + 2 \frac{\Delta t}{\tau_T} \frac{T_{eq} - \mathcal{T}(t)}{\mathcal{T}(t)} \quad \text{and} \quad \alpha_P(t) = 1 - \frac{\Delta t}{\tau_P K} (\mathcal{P}_{eq} - \mathcal{P}(t)),$$

where  $\mathcal{T}$  and  $\mathcal{P}$  are denoting instantaneous values for  $T$  and  $P$ . For  $\alpha_P$ ,  $K \equiv -V \left( \frac{\partial P}{\partial V} \right)_T = 76$  GPa is the bulk modulus for aluminum [5].

For the solid state, the system was equilibrated to  $T_{eq} = 500$  °C and  $P_{eq} = 1$  bar = 0.1 MPa with 25 000 iterations, during  $25\,000 \cdot \Delta t = 25$  ps. The time constants  $\tau_T$  and  $\tau_P$  were set to  $200 \Delta t$  and  $1000 \Delta t$  respectively, since these values gave reasonable convergence to the desired equilibrium. Figure 5 displays the instantaneous temperature and pressure during equilibration. Note, again, that the equilibration time is  $t_{eq} = 25$  ps, and that the temperature and pressure are converging to the set values 500 °C and 0.1 MPa.

During production ( $t > 25$  ps in Figure 5), the average temperature was 496.6 °C and average pressure was 2.1 MPa. Due to the large fluctuations displayed in 5, these values must be considered reasonable.  $T_{avg}$  deviates only 3.4 °C from  $T_{eq}$  while  $P_{avg}$  deviates substantially more from  $P_{eq}$ , but has relatively much larger fluctuations around 100 MPa. See also Figure A.1 in Appendix A which shows the energy-temperature graph for this simulation. This implies that the energy was conserved during the run, assuring valid results.

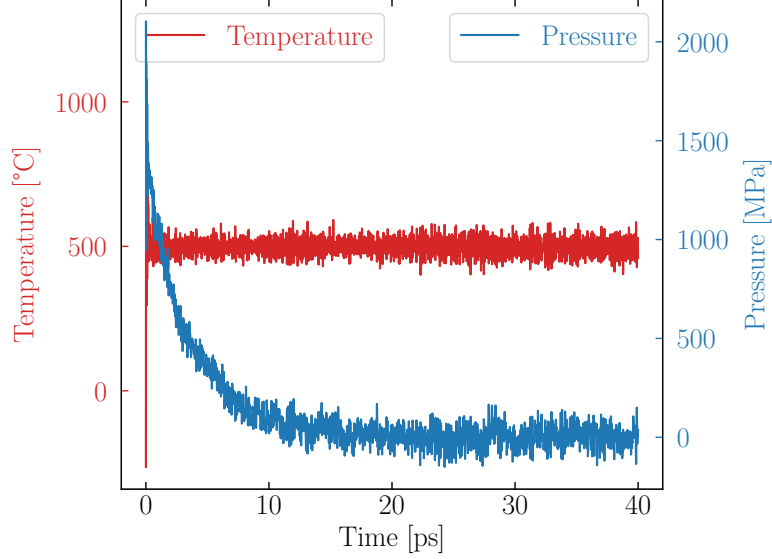


Figure 5: Equilibration of temperature and pressure with equilibration time  $t_{eq} = 25$  ps,  $T_{eq} = 500$  °C and  $P_{eq} = 0.1$  MPa.

In addition to scaling positions and velocities during equilibration, the lattice parameter  $a_0$  was also scaled with  $\alpha_P^{1/3}$  in each iteration. It follows from Figure 6, where the unit cell volume ( $V = a_0^3$ ) is plotted for each time-step, that the unit cell reaches a maximum of  $\sim 70 \text{ \AA}^3$  after roughly 15 ps. After the equilibration, the lattice parameter is hence  $\sim 4.12 \text{ \AA}$ , which is slightly higher than  $a_0 = 4.03 \text{ \AA}$ , obtained at 0 K. This increase in lattice parameter is a consequence of thermal expansion, or in other words, the fact the materials increase their volume with increased temperature.

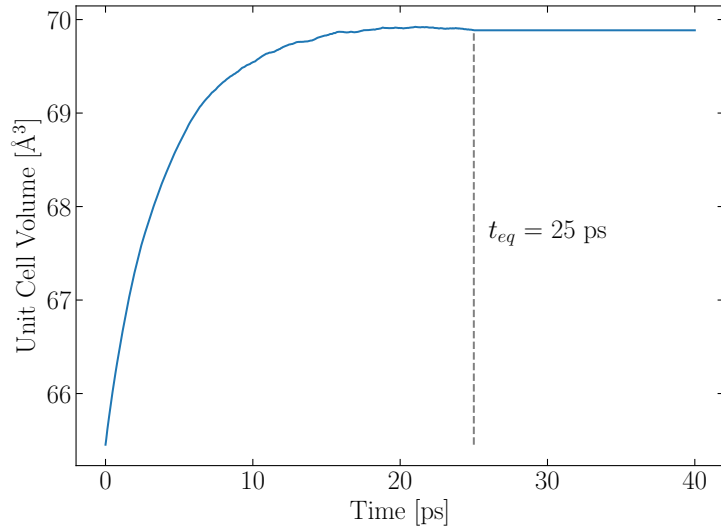


Figure 6: Unit cell volume during the equilibration ( $t < t_{eq} = 25$  ps) and production ( $t > t_{eq}$ ) simulation runs.

Moreover, the trajectories for four chosen atoms were plotted, see Figure 7. When studying individual atoms these four were used throughout the tasks, i.e. similar Figures show the same atoms. Notice how the  $x$ -,  $y$ - and  $z$ -coordinate stay close to their respective initial position for all atoms, showing that the system indeed is in a solid state.

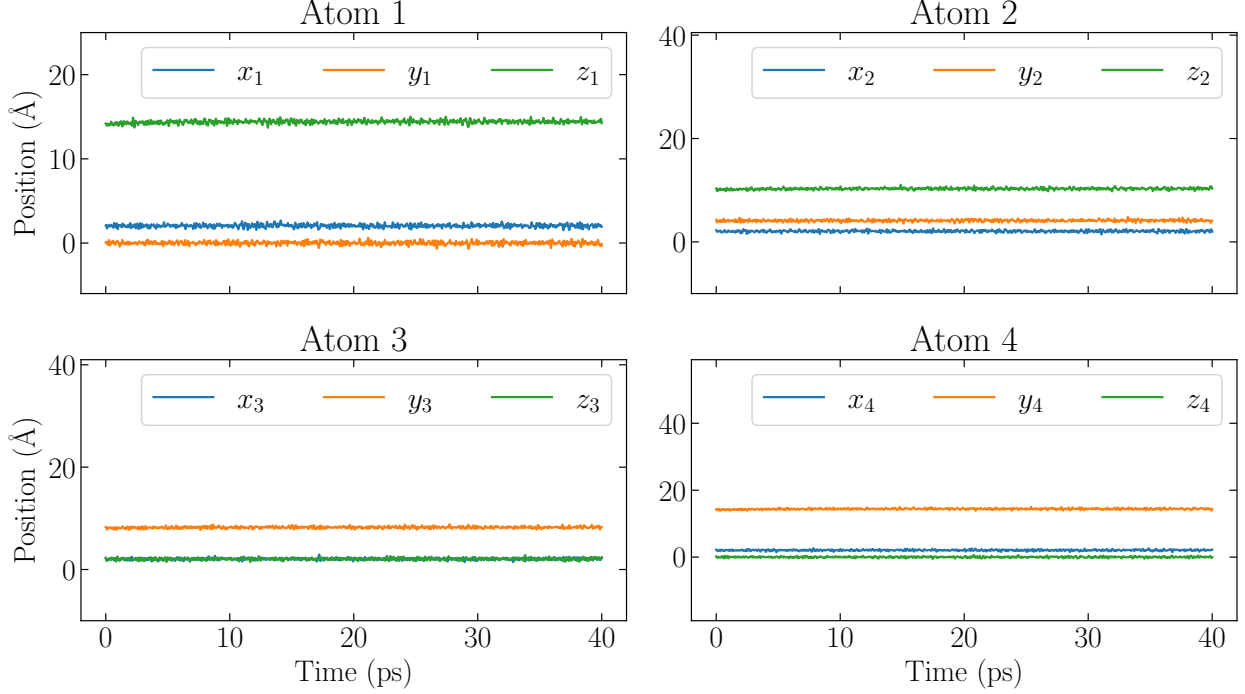


Figure 7: Atomic trajectories:  $x$ -,  $y$ - and  $z$  coordinates, for four atoms as a function of time during the equilibration (25 ps) + production (15 ps) simulation.

## Task 4 – Characteristics of the Melted System

Similar to the previous task, an equilibration was made, however, this time in a liquid state at  $T_{eq} = 700^\circ\text{C}$  and  $P_{eq} = 0.1\text{ MPa}$ . First, the system had to be liquefied, and that was done with an equilibration at  $T = 900^\circ\text{C}$  (substantially higher than the melting point of aluminum  $T_m \approx 660^\circ\text{C}$  [6]) for 50 ps. Secondly, another equilibration at  $T_{eq} = 700^\circ\text{C}$  was run for 30 ps and finally, a production simulation for 20 ps (here, the pressure was kept the same as above). During the production simulation, an average temperature of  $T_{avg} = 699.6^\circ\text{C}$  and average pressure of  $P_{avg} = -7.7\text{ MPa}$  was measured. The temperature value deviates roughly  $0.4^\circ\text{C}$  from  $T_{eq}$ , and does obviously makes sense. On the contrary, the negative pressure value is somewhat unreasonable. However, that could be a consequence of the small system and that  $P_{eq} = 0.1\text{ MPa}$  is set quite low. Moreover, the pressure values in Figure 8 shows larger fluctuations  $\sim \pm 100\text{ MPa}$  that are huge compared to  $P_{eq}$ , which also could contribute to the non-physical negative average pressure. Looking forward, a more sophisticated method for equilibrating pressure may provide a more accurate result.

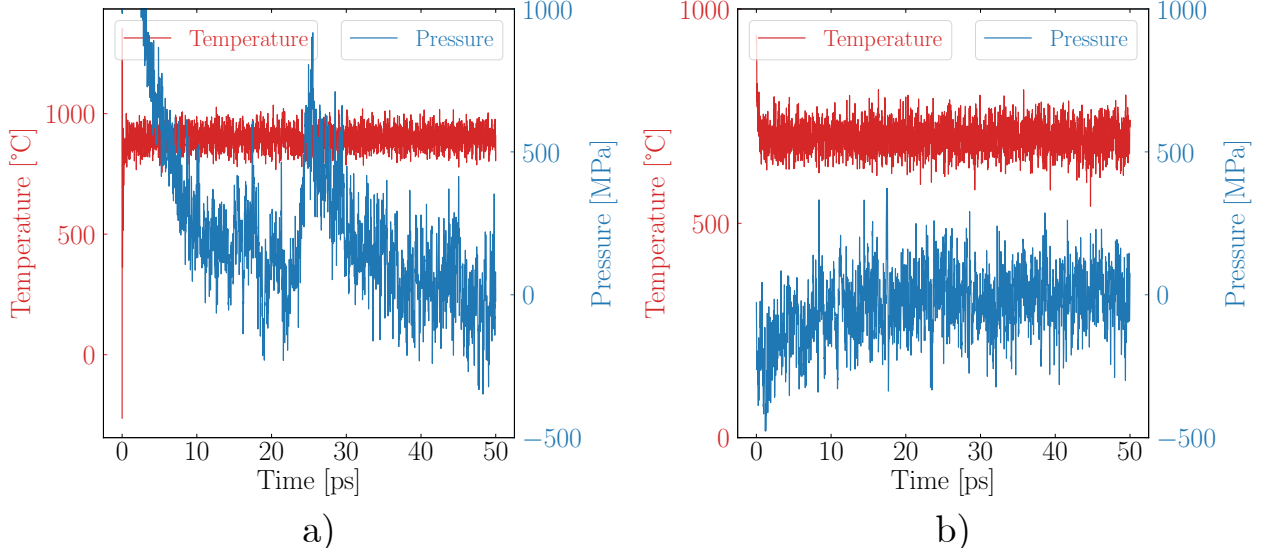


Figure 8: Temperature and pressure during the first long equilibration at  $T = 900^\circ\text{C}$  (a) and during the second equilibration (30 ps) at  $T = 700^\circ\text{C}$  and production (20 ps) simulation (b).

For the liquid state, individual atom trajectories were plotted in Figure 9. By comparing the trajectories in Figure 9 to Figure 7 (note that the  $y$ -axes for each particle have the same limits and that the atoms are the same), it can be concluded that the trajectories deviate substantially from the initial positions, which proves that the system has properly been melted and transitioned into a liquid state.

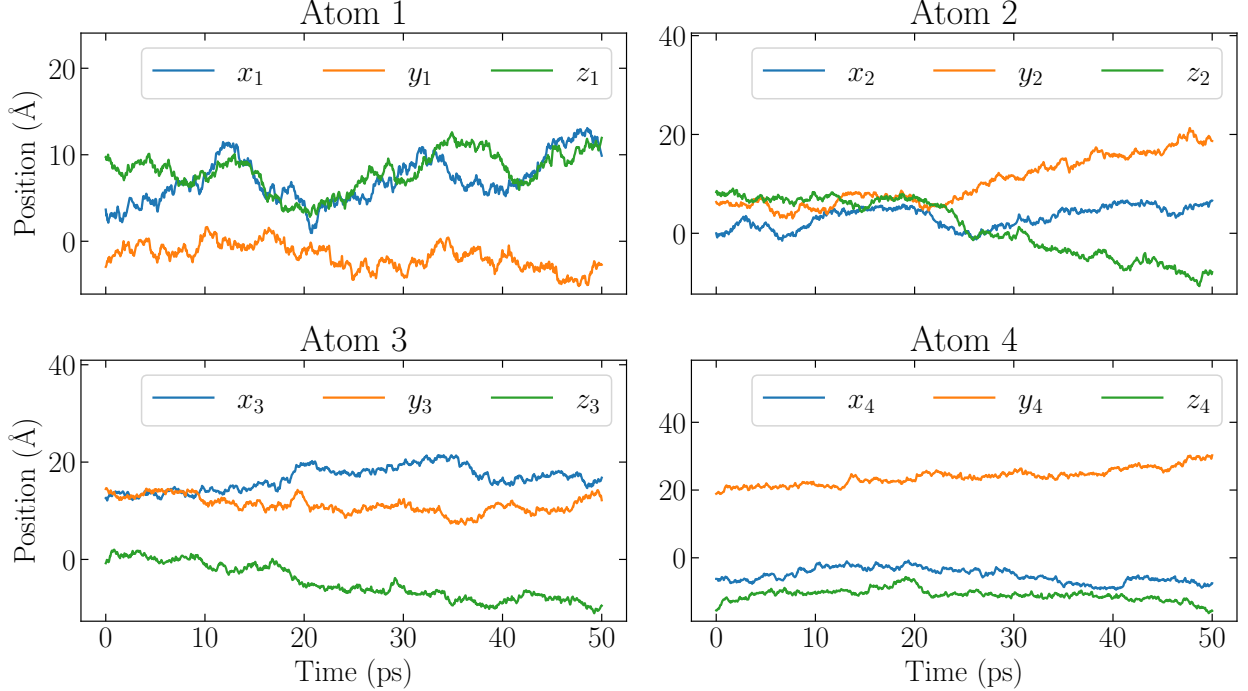


Figure 9: Atomic trajectories:  $x$ -,  $y$ - and  $z$  coordinates, for four atoms as a function of time during the second equilibration (30 ps) + production (20 ps) simulation.



See also Figures A.2, A.3, A.4 and A.5 in Appendix A, showing additional results from the long equilibrium and equilibrium + production simulation described above. During the first long equilibrium run, Figures A.2 and A.3 show the energy, temperature and trajectories of the same four atoms as in Figure 9 above. Here, the phase transition from a solid to liquid state is visible as a jump in energy and deviating trajectories for the atoms, similar to Figure 9. Figures A.4 and A.5 show the energy, temperature and unit cell volume evolution during the equilibrium/production simulation. These show that the energy and volume stay constant during the production, assuring valid results. The fact that the volume is constant is also important in the next task, where the heat capacity at constant volume was computed.

## General Information Tasks 5 – 7

When calculating and studying the static properties of the liquid system in the following tasks, we used the same simulation set-up as in Task 4. Furthermore, the same initial conditions were used as in the previous tasks – i.e. initial velocities set to zero and positions were slightly disturbed as described in Task 2.

### Task 5 – Determining Heat Capacity

In this task, the melted system’s heat capacity at constant volume,  $C_V$ , was calculated using the provided relation

$$C_V = \frac{3Nk_B}{2} \left( 1 - \frac{2}{3Nk_B^2 T^2} \langle (\delta\mathcal{E})^2 \rangle \right)^{-1}, \quad (1)$$

where  $N$  is the number of atoms and  $\delta\mathcal{E} = \mathcal{E} - \langle E \rangle$  the energy fluctuation. Equation 1 was then used with both  $\mathcal{E} = \mathcal{E}_{\text{pot}}$  and  $\mathcal{E} = \mathcal{E}_{\text{kin}}$ . The found  $C_V$  values for both cases are given in Table 1 together with average temperatures and energy fluctuation.

Table 1: Heat capacity ( $C_V$ ), energy fluctuations ( $\delta\mathcal{E}$ ) calculated with  $E_{\text{kin}}$  and  $E_{\text{pot}}$  along with average temperatures ( $T_{\text{avg}}$ ) for the solid and the liquid states.

State	$C_{V,\text{kin}}$ [J/K]	$C_{V,\text{pot}}$ [J/K]	$\delta\mathcal{E}_{\text{kin}}$ [eV]	$\delta\mathcal{E}_{\text{pot}}$ [eV]	$T_{\text{avg}}$ [K]
Solid	$1.2 \times 10^{-20}$	$1.2 \times 10^{-20}$	0.9	0.9	768
Liquid	$9.8 \times 10^{-21}$	$9.8 \times 10^{-21}$	1.2	1.2	972

From the obtained  $C_V$  values in Table 1,  $C_V$  per mol can be calculated accordingly:

$$C_V \left( \frac{N}{N_A} \right)^{-1} = \begin{cases} 1.2 \times 10^{-20} \left( \frac{256}{6.022 \times 10^{23}} \right)^{-1} \approx 28.2 \text{ J}/(\text{mol} \cdot \text{K}), & \text{(solid)} \\ 9.8 \times 10^{-21} \left( \frac{256}{6.022 \times 10^{23}} \right)^{-1} \approx 23.1 \text{ J}/(\text{mol} \cdot \text{K}), & \text{(liquid).} \end{cases} \quad (2)$$

Comparing to tabulated values, the heat capacity for solid aluminum is roughly  $30.0 \text{ J}/(\text{mol} \cdot \text{K})$  [7], which is similar to the obtained value in Equation 2. For the liquid state, on the other hand, the tabulated value is  $31.8 \text{ J}/(\text{mol} \cdot \text{K})$ , which is slightly higher than the obtained value in Table 1. This might be due to the small size of the system and that we use a somewhat simplified model.

## Task 6 – Determining the RDF

In this task, the radial distribution function (RDF) for the liquid state,  $g(r)$ , was calculated. Since the system is isotropic and homogeneous, only the radial component of  $g(r)$  was considered. Obtaining  $g(r)$  was done by counting the distance between all pairs of particles and sorting these into a histogram for every time-step in the *Verlet algorithm*. To achieve this, the function `radial_dist` (provided in the Code Snippet Task 6) was used, where `bdry_dist_between_vectors` calculates distance between position vectors with respect to the periodic boundary conditions, i.e.  $\vec{r}_i - \vec{r}_j = \Delta\vec{r}_{ij} = L - \Delta\vec{r}_{ij}L$ .

### Code Snippet Task 6, Radial Distribution Function

```

1  void
2  radial_dist(double *bins, double **positions, int N_bins, double bin_width, int N,
   ↪ double L)
3  {
4      double r; // Distance between atoms
5      double V = L * L * L; // Volume of the supercell
6      double norm_factor; // Normalization factor
7      for (int i = 0; i < N; i++)
8      {
9          for (int j = 0; j < N; j++)
10         {
11             if (i != j)
12             {
13                 // Calculate the distance between the atoms
14                 r = bdry_dist_between_vectors(positions[i], positions[j], 3, L);
15                 for (int l = 0; l < N_bins; l++)
16                 {
17                     // Check if the distance is within the bin, and increment the
18                     // bin if true
19                     if (l * bin_width < r && r < (l + 1) * bin_width)
20                     {
21                         bins[l] += 1;
22                     }
23                 }
24             }
25         }
26     }
27     for (int l = 0; l < N_bins; l++)
28     {
29         bins[l] /= (N - 1); // Average over the number of atoms
30         norm_factor = (N - 1) * 4 * M_PI * (3 * pow((l + 1), 2) - 3 * l + 1) *
   ↪ pow(bin_width, 3) / 3 / V;
31         bins[l] /= norm_factor; // Apply the scale factor
32     }
33 }
```

By choosing a bin-width  $\Delta r = 0.05$ , the `radial_dist` function calculates the average number of particles that lies within a distance  $r \in [(k - 1)\Delta r, k\Delta r]$  from a given particle, denoted  $\langle N_k \rangle$ . For each time a certain distance lies in this interval for a certain  $k$ , bin  $k$  is increased by one. In the end, each bin is divided by  $(N - 1)$  to get the average number of particles, see the code snippet above. Then,  $g(r)$  can be obtained by dividing bin  $k$  by the factor

$$N_k^{ideal} = \frac{(N - 1) 4\pi}{V} (3k^2 - 3k + 1) \Delta r^3, \quad (3)$$

where  $N$  again is the total number of particles and  $V$  the volume of the supercell (see code snippet above). In essence, this whole term describes the average number of atoms in a (spherical) shell assuming a random distribution. Lastly, the term  $\frac{N-1}{V}$  assures that  $g(r)$  converges to 1 for large  $r$ .

These calculations were then made for the atom positions at each time-step in the *Verlet algorithm* and the final  $g(r)$  was obtained through a time average over all time-step's individual bin counts. The total number of bins was defined as  $N_{bins} = L/(2\Delta r)$ , so that  $1 \leq k \leq N_{bins}$ . This definition of  $N_{bins}$  assures that no specific distances are double-counted, respecting the periodicity of the lattice structure.

The RDF for the liquid state of the studied lattice is shown in Figure 10 below. This is similar to previous findings by Song and Morris [8]. The function describes at what distances  $r$  from a given atom another atom is most likely to be found, where peaks means high probability. Notice how it starts to converge to 1 when  $r$  becomes larger, as expected from our methodology. However, this is also an important property of the RDF since this means that for large distances in the lattice, the atoms become more and more uniformly distributed, with no correlation. For a solid, we would not expect this to happen since the atom positions are more or less arranged to the FCC structure.

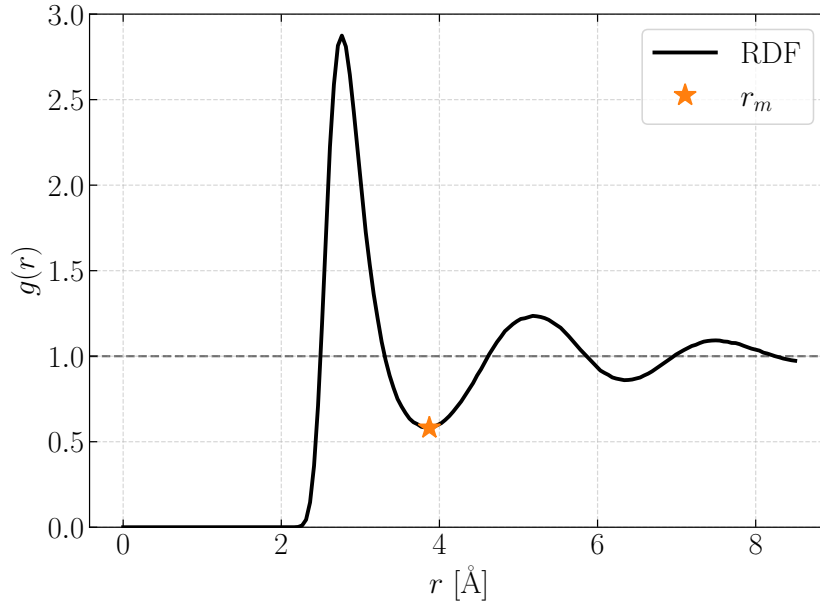


Figure 10: The radial distribution function  $g(r)$  as a function of distance between the atom pairs in the lattice structure. The orange star denotes  $r_m$ , which was used to calculate the coordination number  $I(r_m)$ .

Moreover, by integrating  $g(r)$  according to

$$I(r_m) = \frac{N}{V} \int_0^{r_m} g(r) 4\pi r^2 dr, \quad (4)$$

the coordination number  $I$  was obtained. This corresponds to the number of atoms whose distances lies in the range  $r < r_m$  from a given atom in the lattice. In our case,  $r_m$  was set to the  $r$ -value of the first local minimum of  $g(r)$ , which corresponds to the orange star in Figure 10,  $r_m \approx 3.87 \text{ Å}$ . Using Equation 4, a coordination number of  $I(r_m) \approx 12.11$  was obtained. For a FCC lattice, this value is slightly higher than the theoretical value of 12.

Though, we consider the liquid state, where the FCC structure should have erupted. However, this result could still be reasonable since the coordination number here only takes the short-range perspective in account, where our sample could probably be approximated to still be in a FCC configuration. Lastly, a finer spaced grid (e.g. more bins) should potentially also give a slightly more accurate value of  $I(r_m)$ .

## Task 7 – Determining the Static Structure Factor

Lastly, we determined the static structure factor (SSF),  $S(\mathbf{q})$ , for the liquid aluminum configuration. This was done using the provided formula

$$S(\mathbf{q}) = \frac{1}{N} \left\langle \sum_{i=1}^N \sum_{j=1}^N e^{-i\mathbf{q} \cdot (\mathbf{r}_i(t) - \mathbf{r}_j(t))} \right\rangle = \frac{1}{N} \left\langle \left( \sum_{i=1}^N \cos(\mathbf{q} \cdot \mathbf{r}_i(t)) \right)^2 + \left( \sum_{i=1}^N \sin(\mathbf{q} \cdot \mathbf{r}_i(t)) \right)^2 \right\rangle. \quad (5)$$

Here  $\mathbf{r}_i(t)$  is every atom's position at time  $t$  and  $\mathbf{q}$  is a 3-dimensional grid consistent with the periodic boundary conditions, i.e.  $\mathbf{q} = \frac{2\pi}{L}(n_x, n_y, n_z)$ , where  $n_x$ ,  $n_y$  and  $n_z$  are integers  $\in [-N_{max}, N_{max}]$ . Also notice that the equivalent rewriting of the first expression in Equation 5 will save us  $N$  computational time units. In order to create the  $\mathbf{q}$ -grid, a function was implemented in C, see the code snippet below.

### Code snippet task 7 Initialize Grid

```

1  double ****
2  init_grid(int N_max, double L)
3  {
4      int N = 2*N_max + 1;
5      double ****grid = (double ****)malloc(N * sizeof(double ****));
6      for (int i = 0; i < N; i++)
7      {
8          grid[i] = (double ***)malloc(N * sizeof(double **));
9          for (int j = 0; j < N; j++)
10         {
11             grid[i][j] = (double **)malloc(N * sizeof(double *));
12             for (int k = 0; k < N; k++)
13             {
14                 grid[i][j][k] = (double *)malloc(3 * sizeof(double));
15                 grid[i][j][k][0] = 2 * M_PI / L * (i - N_max);
16                 grid[i][j][k][1] = 2 * M_PI / L * (j - N_max);
17                 grid[i][j][k][2] = 2 * M_PI / L * (k - N_max);
18             }
19         }
20     }
21     return grid;
22 }
```

This grid was then used to calculate the sums in Equation at every time-step in the *Verlet algorithm*. In order to obtain a plottable 1-dimensional version of the SSF, a spherical average was performed by considering only the length of the  $\mathbf{q}$ -vector, i.e.  $q = |\mathbf{q}|$ . The calculated values at every time-step for each  $\mathbf{q}(q)$  was sorted into a number of bins,  $N_{bins}$  with a certain bin width  $\Delta q$ , see again the code snippet below.

### Code snippet task 7, Spherical Average

```

1 void
2 spherical_avg(double ****grid, double *Sq, int n, int n_bins, double bin_width)
3 {
4     double *S_avg = calloc(n_bins, sizeof(double));
5     int *counts = calloc(n_bins, sizeof(double));
6     for (int i = 0; i < n; i++)
7     {
8         for (int j = 0; j < n; j++)
9         {
10            for (int k = 0; k < n; k++)
11            {
12                double q = sqrt(grid[i][j][k][0] * grid[i][j][k][0] +
13                               grid[i][j][k][1] * grid[i][j][k][1] +
14                               grid[i][j][k][2] * grid[i][j][k][2]);
15                int bin = (int)(q / bin_width);
16                if (bin == n_bins){
17                    bin = n_bins - 1;
18                }
19                S_avg[bin] += Sq[i * n * n + j * n + k];
20                counts[bin]++;
21            }
22        }
23    }
24 }

```

When computing the SSF, we used a value of  $N_{max} = 20$ , resulting in a  $q_{max} = |\mathbf{q}|_{max} = \sqrt{3} \frac{2\pi N_{max}}{L}$ , where  $L$  again is the length of the supercell. Moreover, a number of  $N_{bins} = 300$  was used. However, this setup was a bit time-consuming and probably not ideal for our personal laptops' health, but what will a simple man not do for science. The result is displayed in Figure 11 below, which is consistent with previous findings by [9]. Notice also how the SSF converges to 1 as  $q$  increases. This is, as in the case of  $g(r)$ , expected since this means that the liquid system lacks order.

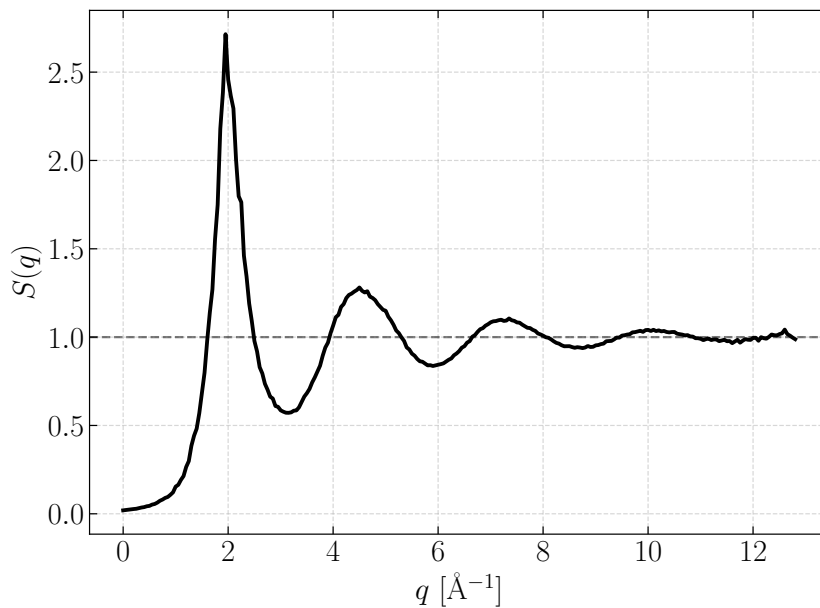


Figure 11: The static structure factor  $S(q)$  for liquid aluminum at  $T = 972$  K.

## Conclusion

To summarize, the dynamics of an aluminum lattice with 256 atoms have been simulated using C. The system was in general run with  $\Delta t = 0.001$  ps, ensuring stable, non-diverging, energies and temperature. Comparing the solid and the liquid system, the liquid system has trajectories that deviate from the initial positions, which signifies that the model has accurately captured the phase transition. In addition, the shape of the radial distribution function  $g(r)$  and static structure factor  $S(q)$ , both converging to 1 for large  $r / q$ , also indicates that the system is in the liquid state. Although this, the model tend to give a slightly lower  $C_V$  when compared to the tabulated values, especially for the liquid state. This suggests that the model still could be improved upon.

From our own perspective, we found it fascinating how well the model is able to capture the system dynamics, especially the phase transition from solid to liquid. It is obvious that the art of studying molecular dynamics has evolved significantly since the 1950. We bet that Fermi, Pasta, Ulam and Tsingou would have been thrilled to see that such an accurate description of the dynamics can be obtained using our personal laptops.

## References

- [1] Wikipedia. (), [Online]. Available: [https://en.wikipedia.org/wiki/Fermi%E2%80%9993Pasta%E2%80%9393Ulam%E2%80%9393Tsingou\\_problem](https://en.wikipedia.org/wiki/Fermi%E2%80%9993Pasta%E2%80%9393Ulam%E2%80%9393Tsingou_problem) (visited on 26/11/2024).
- [2] Z. Fan, Y. Wang, P. Ying *et al.*, “Gpumd: A package for constructing accurate machine-learned potentials and performing highly efficient atomistic simulations”, *The Journal of Chemical Physics*, vol. 157, no. 11, p. 114 801, Sep. 2022, ISSN: 0021-9606. DOI: [10.1063/5.0106617](https://doi.org/10.1063/5.0106617). [Online]. Available: <https://doi.org/10.1063/5.0106617>.
- [3] C. Nordling and J. Österman, *Physics Handbook: for Science and Engineering*, 9th. Lund, Sweden: Studentlitteratur, 2020, ISBN: 978-91-44-12806-1.
- [4] D. Belashchenko, A. Vorotyagin and B. Gelchinski, “Computer simulation of aluminum in the high-pressure range”, *High Temperature*, vol. 49, Oct. 2011. DOI: [10.1134/S0018151X11050038](https://doi.org/10.1134/S0018151X11050038).
- [5] Periodictable.com. “Technical data for aluminum”. (), [Online]. Available: <https://periodictable.com/Elements/013/data.html> (visited on 18/12/2024).
- [6] Thyssenkrupp Materials. “Melting point of aluminium”. (), [Online]. Available: <https://www.thyssenkrupp-materials.co.uk/melting-point-of-aluminium> (visited on 04/12/2024).
- [7] National Institute of Standards and Technology. “NIST chemistry webbook, srd 69, aluminum”. (), [Online]. Available: <https://webbook.nist.gov/cgi/cbook.cgi?ID=C7429905&Mask=2&Type=JANAFL&Plot=on#JANAFL> (visited on 27/11/2024).
- [8] X. Song and J. R. Morris, “Accurate method to calculate liquid and solid free energies for embedded atom potentials”, *Phys. Rev. B*, vol. 67, p. 092 203, 9 Mar. 2003. DOI: [10.1103/PhysRevB.67.092203](https://doi.org/10.1103/PhysRevB.67.092203). [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRevB.67.092203>.
- [9] D. J. González, L. E. González, J. M. López and M. J. Stott, “Dynamical properties of liquid al near melting: An orbital-free molecular dynamics study”, *Phys. Rev. B*, vol. 65, p. 184 201, 18 Apr. 2002. DOI: [10.1103/PhysRevB.65.184201](https://doi.org/10.1103/PhysRevB.65.184201). [Online]. Available: <https://link.aps.org/doi/10.1103/PhysRevB.65.184201>.

# Appendix

## A Additional Results

Here we present additional results that was not explicitly asked for in the assignment.

### Task 3

Figure A.1 below shows the energy-temperature plot for the solid system simulation. Note that the total energy is conserved during the production simulation.

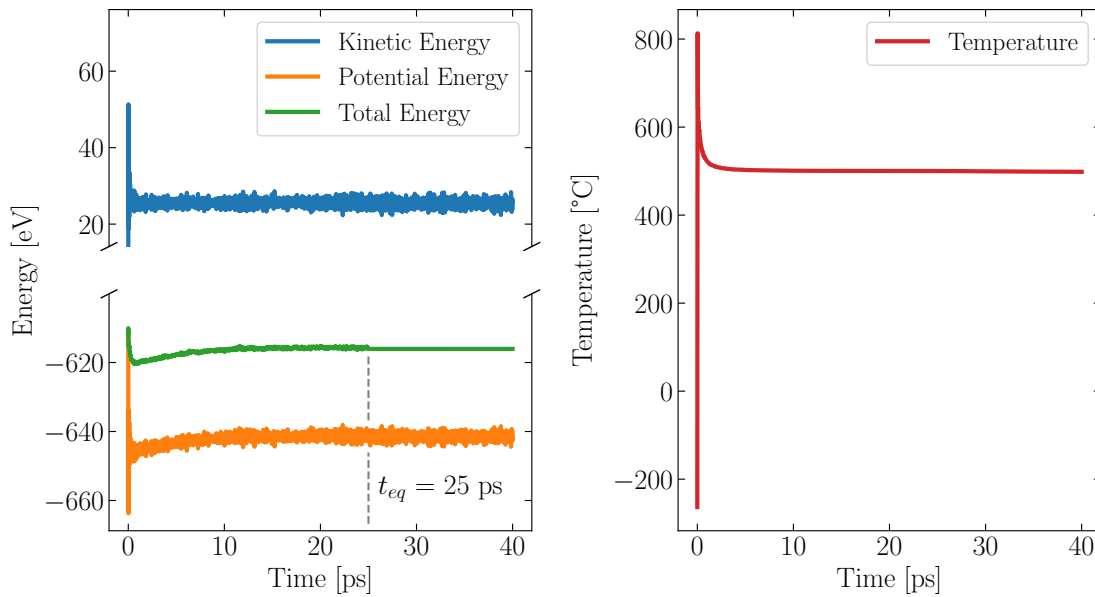


Figure A.1: The plot to the left contain instantaneous kinetic, potential and total energy for the solid system at each time step. The right plot displays the average temperature throughout the simulation. Both plots are for both equilibrium (25 ps) and production (15 ps).

### Task 4

#### Warm-up – Melting the system

Figure A.2 shows the energy-temperature plot during the warm-up phase run for 50 ps in order to melt the system. Figure A.3 shows the trajectories for 4 atoms (the same atoms as in Figures 7 and 9) during the same simulation. Note that the phase transition actually can be observed in both the energy plot (as the sudden jump) and the trajectory plot, as the atoms start to deviate more from their initial positions.



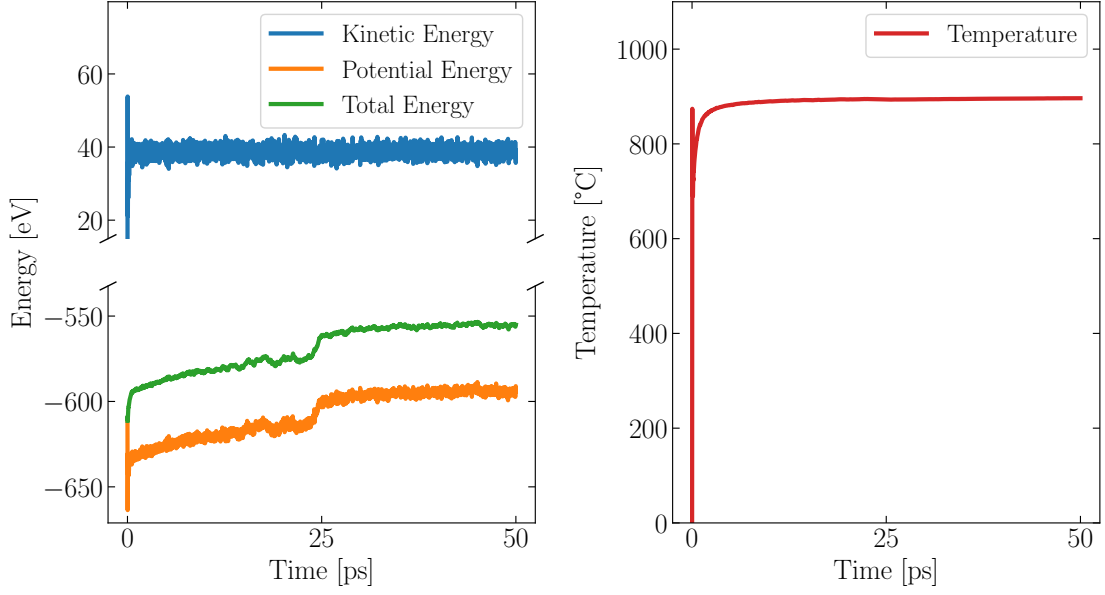


Figure A.2: The plot to the left contain instantaneous kinetic, potential and total energy for the liquid system during the warm-up phase at each time step. The right plot displays the average temperature throughout the simulation.

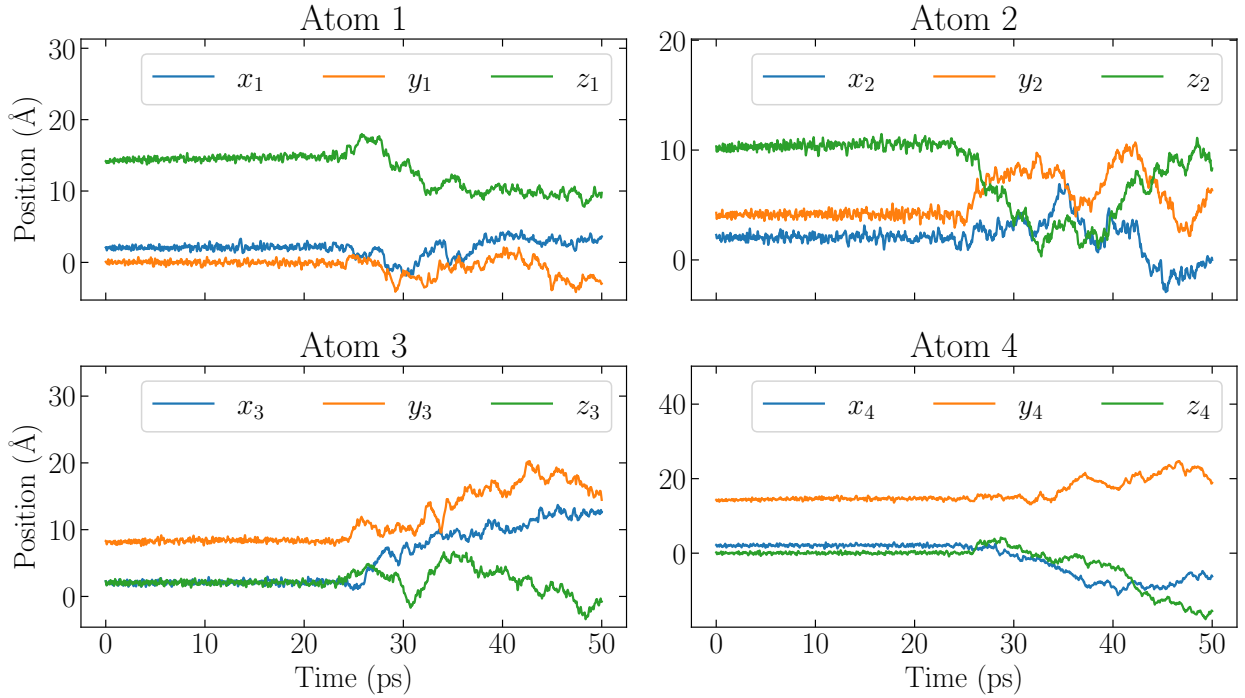


Figure A.3: Atomic trajectories:  $x$ -,  $y$ - and  $z$  coordinates, for four atoms as a function of time.

### Production simulation

Figure A.4 shows the energy-temperature plot during the production run for 50 ps with 30 ps as equilibrium time. Figure A.5 shows the unit cell volume evolution during the simulation.

Note that the energy as well as the volume is conserved after the equilibrium steps.

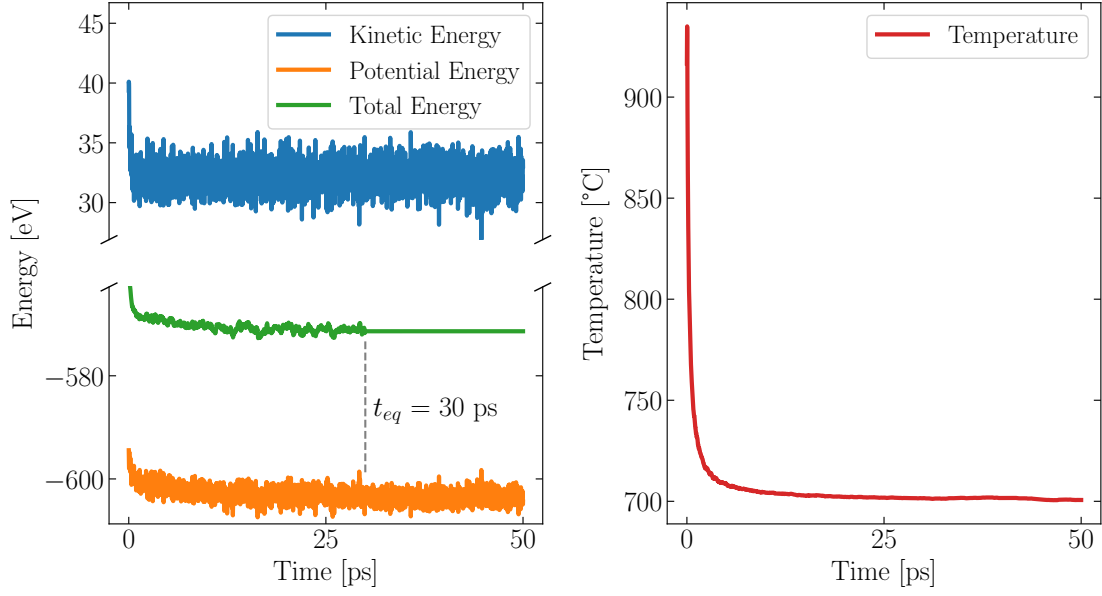


Figure A.4: The plot to the left contain instantaneous kinetic, potential and total energy for the liquid system during the production phase at each time step. The right plot displays the average temperature throughout the simulation.

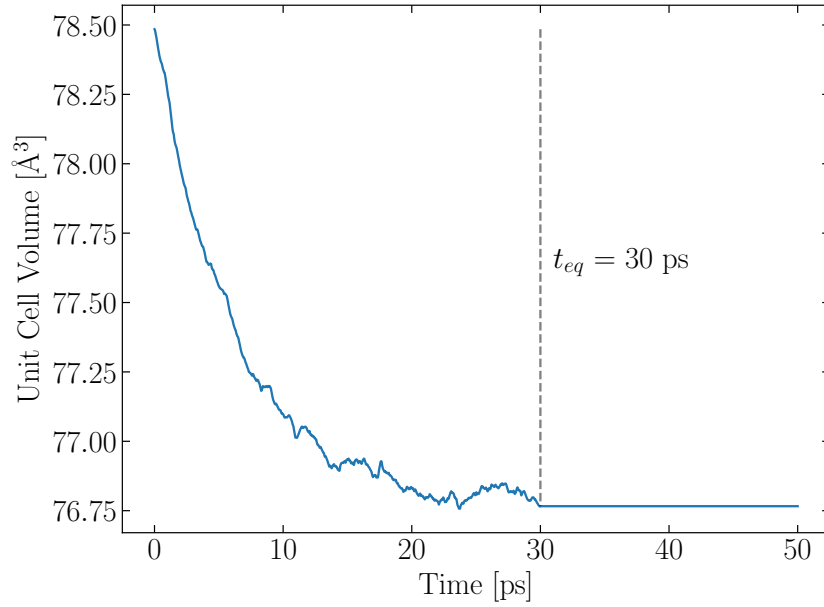


Figure A.5: Unit cell volume during the equilibration ( $t < t_{eq} = 30$  ps) and production ( $t > t_{eq}$ ) simulation runs.

## B Source code in C

### B.1 run.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  #include "lattice.h"
5  #include "potential.h"
6  #include "tools.h"
7  #include "run.h"
8  #include <gsl/gsl_rng.h>
9  #include <gsl/gsl_randist.h>
10
11 int
12 run(int argc, char *argv[])
13 {
14     if (argc < 2) {
15         fprintf(stderr, "Usage: %s <delta_t>\n", argv[0]);
16         return 1;
17     }
18     // ----- //
19     // ***** Constants ***** //
20     // ----- //
21     int N = 256; // Number of atoms
22     int cell_length = 4; // Number of unit cells in each direction
23     double k_B = 8.617333262145 * 1e-5; // Boltzmann constant [eV/K]
24     double m = 27.0 / 9649; // Aluminium mass [eV ps2/Å2]
25     double beta = 1. / (76e3); // Bulk Modulus inverse [1 / MPa]
26
27     // ----- //
28     // ***** Task 1 ***** //
29     // ----- //
30     // Lattice parameters
31     double a0[] = {4.0, 4.005, 4.01, 4.015, 4.02, 4.025, 4.03, 4.035, 4.04, 4.045,
32                   4.05, 4.055, 4.06, 4.065, 4.07, 4.075, 4.08};
33
34     // Energy file
35     char *filename = "data/task_1/energies.csv";
36     FILE *fp = fopen(filename, "w");
37
38     // Iterate over the lattice parameters and calculate the energy
39     for (int i = 0; i < 17; i++)
40     {
41         // Positions vector
42         double **pos = create_2D_array(N, 3);
43         // Initialize the fcc lattice
44         init_fcc(pos, 4, a0[i]);
45         // Calculate the energy
46         double energy = get_energy_AL(pos, a0[i] * cell_length, N);
47         // Write the energy to the file
48         fprintf(fp, "%f, %f\n", a0[i], energy);
49         // Free the memory
50         destroy_2D_array(pos);
51     }
52
53     fclose(fp);
```

```

54
55 // ----- //
56 // ***** Initialize Task 2 & 3 & 4 ***** //
57 // ----- //
58 int t_max = 50; // The simulation time [ps]
59 const double delta_t = atof(argv[1]); // The time step [ps]
60 int its = (int)(t_max / delta_t); // Number of iterations
61
62 double **positions = create_2D_array(N, 3); // Atom positions
63 double **forces = create_2D_array(N, 3); // Forces
64 double **velocities = create_2D_array(N, 3); // Velocities
65 // Initialize the velocities to 0
66 for (int i = 0; i < N; i++)
67 {
68     for (int j = 0; j < 3; j++)
69     {
70         velocities[i][j] = 0.0;
71     }
72 }
73
74 // Initialize the random number generator
75 gsl_rng *r = init_gsl_rng(42);
76
77 double a_0 = 4.03; // Lattice constant at 0 K [Å]
78 init_fcc(positions, 4, a_0); // Initialize lattice
79 for (int i = 0; i < N; i++)
80 {
81     double rng = 0.935 + 0.13 * gsl_rng_uniform(r); // Disturbance
82     // Add the disturbance to all initial atom positions
83     addition_with_constant(positions[i], positions[i], a_0 * (1 - rng), 3);
84 }
85
86 // ----- //
87 // ***** Task 2 ***** //
88 // ----- //
89 int its_eq = 0; // Number of iterations for equilibration, 0 for task 2
90
91 char filename[50];
92 // Format the filename with delta_t, its and its_eq
93 sprintf(filename, "data/task_2/data_%.3f%i%i.csv", delta_t, its, its_eq);
94 FILE *fp = fopen(filename, "w");
95 // Write the header
96 fprintf(fp, "its, t_max, delta_t, its_eq, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i, %i\n",
97     ↪ its, t_max, delta_t, its_eq, 0, 0, 0, 0);
98 fprintf(fp, "E_kin [eV], E_pot [eV], E_tot [eV], <T> [K], T [K], <P> [MPa], P [MPa],
99     ↪ a [Å]\n");
100
101 // Perform the Verlet algorithm
102 a_0 = verlet(positions, velocities, forces, its, its_eq, delta_t, m, k_B, beta, a_0,
103     ↪ N, cell_length, 0., 0., fp, NULL, NULL, NULL);
104
105 // ----- //
106 // ***** Task 3 ***** //
107 // ----- //
108 double T_eq = 500. + 273.15; // Temperature at equilibrium [K]
109 double P_eq = 0.1; // Pressure at equilibrium [MPa]
110 int its_eq = 25000; // Number of iterations for equilibration

```

```

108     char filename[50];
109     // Format the data filename with delta_t, its and its_eq
110     sprintf(filename, "data/task_3/data_%.3f_%i_%i.csv", delta_t, its, its_eq);
111     FILE *fp = fopen(filename, "w");
112     // Write the data header
113     fprintf(fp, "its, t_max, delta_t, its_eq, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i, %i\n",
114             its, t_max, delta_t, its_eq, 0, 0, 0, 0);
115     fprintf(fp, "E_kin [eV], E_pot [eV], E_tot [eV], <T> [K], T [K], <P> [MPa], P [MPa],
116             a [Å]\n");
117
118     // Format the trajectory filename with delta_t, its and its_eq
119     sprintf(filename, "data/task_3/trajs_%.3f_%i_%i.csv", delta_t, its, its_eq);
120     FILE *fp_traj = fopen(filename, "w");
121     // Write the trajectory header
122     fprintf(fp_traj, "its, t_max, delta_t, its_eq, -, -, -, -, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i, %i, %i\n",
123             its, t_max, delta_t, its_eq, 0, 0, 0, 0, 0, 0, 0, 0);
124     fprintf(fp_traj, "x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3, x_4, y_4, z_4\n");
125
126     // Perform the Verlet algorithm
127     a_0 = verlet(positions, velocities, forces, its, its_eq, delta_t, m, k_B, beta, a_0,
128             N, cell_length, T_eq, P_eq, fp, fp_traj, NULL, NULL);
129
130     // ----- //
131     // ***** Task 4 ***** //
132     // ----- //
133     int its_eq = 50000; // Number of iterations for first equilibration phase, task 4
134     double T_eq = 900. + 273.15; // Temperature at first equilibrium [K]
135     double P_eq = 0.1; // Pressure at first equilibrium [MPa]
136
137     char filename[50];
138     // Format the data filename with delta_t, its and its_eq
139     sprintf(filename, "data/task_4/data_%.3f_%i_%i.csv", delta_t, its, its_eq);
140     FILE *fp_1 = fopen(filename, "w");
141     // Write the data header
142     fprintf(fp_1, "its, t_max, delta_t, its_eq, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i, %i, %i\n",
143             its, t_max, delta_t, its_eq, 0, 0, 0, 0);
144     fprintf(fp_1, "E_kin [eV], E_pot [eV], E_tot [eV], <T> [K], T [K], <P> [MPa], P [MPa], a [Å]\n");
145
146     // Format the trajectory filename with delta_t, its and its_eq
147     sprintf(filename, "data/task_4/trajs_%.3f_%i_%i.csv", delta_t, its, its_eq);
148     FILE *fp_2 = fopen(filename, "w");
149     // Write the trajectory header
150     fprintf(fp_2, "its, t_max, delta_t, its_eq, -, -, -, -, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i, %i, %i\n",
151             its, t_max, delta_t, its_eq, 0, 0, 0, 0, 0, 0, 0, 0);
152     fprintf(fp_2, "x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3, x_4, y_4, z_4\n");
153
154     // Perform the Verlet algorithm for the first equilibration phase
155     a_0 = verlet(positions, velocities, forces, its, its_eq, delta_t, m, k_B, beta, a_0,
156             N, cell_length, T_eq, P_eq, fp_1, fp_2, NULL, NULL);
157
158     t_max = 50; // The simulation time for the second phase [ps]
159     its = (int)(t_max / delta_t); // Number of iterations for the second phase
160     its_eq = 30000; // Number of iterations for the second equilibration phase

```

```

155 T_eq = 700. + 273.15; // Temperature at the second equilibrium [K]
156
157 // Format the data filename with delta_t, its and its_eq
158 sprintf(filename, "data/task_4/data_%.3f_%i_%i.csv", delta_t, its, its_eq);
159 FILE *fp_3 = fopen(filename, "w");
160 // Write the data header
161 fprintf(fp_3, "its, t_max, delta_t, its_eq, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i,\n",
    ↪ %i\n", its, t_max, delta_t, its_eq, 0, 0, 0, 0);
162 fprintf(fp_3, "E_kin [eV], E_pot [eV], E_tot [eV], <T> [K], T [K], <P> [MPa], P\n",
    ↪ [MPa], a [Å]\n");
163
164 // Format the trajectory filename with delta_t, its and its_eq
165 sprintf(filename, "data/task_4/trajs_%.3f_%i_%i.csv", delta_t, its, its_eq);
166 FILE *fp_4 = fopen(filename, "w");
167 // Write the trajectory header
168 fprintf(fp_4, "its, t_max [ps], delta_t [ps], its_eq, T_eq [K], P_eq [MPa], -, -, -, \n",
    ↪ -, -, -, -\n%i, %i, %f, %i, %f, %f, %i, %i, %i, %i, %i, %i\n", its, t_max, delta_t,
    ↪ its_eq, T_eq, P_eq, 0, 0, 0, 0, 0, 0, 0, 0);
169 fprintf(fp_4, "x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3, x_4, y_4, z_4\n");
170
171 // Format the radial distribution filename with delta_t, its and its_eq
172 sprintf(filename, "data/task_4/rdist_%.3f_%i_%i.csv", delta_t, its, its_eq);
173 FILE *fp_rdist = fopen(filename, "w");
174 // Write the radial distribution header
175 fprintf(fp_rdist, "its, its_eq, delta_t [ps], a_0 [Å]\n");
176
177 // Format the structure factor filename with delta_t, its and its_eq
178 sprintf(filename, "data/task_4/sfact_%.3f_%i_%i.csv", delta_t, its, its_eq);
179 FILE *fp_sfact = fopen(filename, "a");
180 // Write the structure factor header
181 fprintf(fp_sfact, "its, its_eq, delta_t [ps], a_0 [Å]\n");
182
183 // Perform the Verlet algorithm for the second phase
184 a_0 = verlet(positions, velocities, forces, its, its_eq, delta_t, m, k_B, beta, a_0,
    ↪ N, cell_length, T_eq, P_eq, fp_3, fp_4, fp_rdist, fp_sfact);
185
186 // Close files and free memory
187 fclose(fp);
188 fclose(fp_traj);
189 fclose(fp_1);
190 fclose(fp_2);
191 fclose(fp_3);
192 fclose(fp_4);
193 fclose(fp_rdist);
194 fclose(fp_sfact);
195 gsl_rng_free(r);
196 destroy_2D_array(positions);
197 destroy_2D_array(forces);
198 destroy_2D_array(velocities);
199
200 return 0;
201 }
202
203 gsl_rng *
204 init_gsl_rng(int seed){
205     const gsl_rng_type * T;
206     gsl_rng * r;

```

```

207     gsl_rng_env_setup();
208     T = gsl_rng_default; // default random number generator
209     r = gsl_rng_alloc(T); // allocate memory for the random number generator
210
211     if (!r) {
212         fprintf(stderr, "Error: Could not allocate memory for RNG.\n");
213         exit(EXIT_FAILURE); // Exit if allocation fails
214     }
215
216     // Set the seed
217     gsl_rng_set(r, seed);
218
219     return r;
220 }
221
222 double
223 verlet(double **positions, double **velocities, double **forces, int its, int its_eq,
224         double delta_t, double m, double k_B, double beta, double a_0, int N,
225         int cell_length, double T_eq, double P_eq, FILE *fp, FILE *fp_traj,
226         FILE *fp_rdist, FILE *fp_sfact)
227 {
228     double tau_T = delta_t * 200; // Time constant for temperature [ps]
229     double tau_P = delta_t * 1000; // Time constant for pressure [ps]
230     double alpha_T; // Scaling factor for temperature
231     double alpha_P; // Scaling factor for pressure
232     double E_pot = 0.0; // Potential energy (instantaneous)
233     double E_kin = 0.0; // Kinetic energy (instantaneous)
234     double virial = 0.0; // Virial
235     double *P = malloc(its * sizeof(double)); // Instantaneous pressure
236     double *T = malloc(its * sizeof(double)); // Instantaneous temperature
237     double T_avg = 0.0; // Average temperature
238     double P_avg = 0.0; // Average pressure
239
240     // Calculate the initial forces, potential energy and virial
241     calculate(&E_pot, &virial, forces, positions, a_0 * cell_length, N);
242     for (unsigned int i = 0; i < its; i++)
243     {
244         E_kin = 0.0;
245         // Perform the first half step
246         for (unsigned int j = 0; j < N; j++)
247         {
248             for (unsigned int k = 0; k < 3; k++)
249             {
250                 velocities[j][k] += 0.5 * delta_t * forces[j][k] / m;
251                 positions[j][k] += delta_t * velocities[j][k];
252             }
253         }
254         // Calculate new accelerations
255         calculate(&E_pot, &virial, forces, positions, a_0 * cell_length, N);
256         // Perform the second half step
257         for (unsigned int j = 0; j < N; j++)
258         {
259             for (unsigned int k = 0; k < 3; k++)
260             {
261                 velocities[j][k] += 0.5 * delta_t * forces[j][k] / m;
262             }
263             E_kin += 0.5 * m * vector_norm(velocities[j], 3) * vector_norm(velocities[j],
↪ 3);

```

```

264     }
265
266     T[i] = 2.0 / 3.0 / k_B / N * E_kin; // Calculate the instantaneous temperature
    ↪ [K]
267     T_avg = average(T, i+1); // Calculate the average temperature [K]
268     P[i] = 1 / (3 * 64 * pow(a_0, 3)) * (E_kin + virial) / 6.2415 * 1e6; // Calculate
    ↪ the instantaneous pressure [MPa]
269     P_avg = average(P, i+1); // Calculate the average pressure [MPa]
270
271     // Scale the temperature and pressure if the equilibration phase is not over
272     if (i < its_eq)
273     {
274         // Calculate the scaling factors
275         alpha_T = 1 + 2 * delta_t * (T_eq - T[i]) / (T[i] * tau_T);
276         alpha_P = 1 - beta * delta_t * (P_eq - P[i]) / tau_P;
277
278         // Scale the lattice constant
279         a_0 = a_0 * pow(alpha_P, 1. / 3.);
280
281         // Scale the temperature and velocities
282         for (unsigned int j = 0; j < N; j++)
283         {
284             multiplication_with_constant(positions[j], positions[j], pow(alpha_P, 1.
    ↪ / 3.), 3);
285             multiplication_with_constant(velocities[j], velocities[j], sqrt(alpha_T),
    ↪ 3);
286         }
287     }
288     // Write the data (energy, temperature, pressure and lattice constant) to the
    ↪ data file
289     if (fp != NULL)
290     {
291         fprintf(fp, "%f, %f, %f, %f, %f, %f, %f, %f\n", E_kin, E_pot, E_kin + E_pot,
    ↪ T_avg, T[i], P_avg, P[i], a_0);
292     }
293     // Write the trajectory data to the trajectory file
294     if (fp_traj != NULL)
295     {
296         fprintf(fp_traj, "%f, %f, %f, %f, %f, %f, %f, %f, %f, %f, %f, %f\n",
297             positions[15][0], positions[15][1], positions[15][2],
298             positions[27][0], positions[27][1], positions[27][2],
299             positions[35][0], positions[35][1], positions[35][2],
300             positions[49][0], positions[49][1], positions[49][2]);
301     }
302     // Calculate the radial distribution and write it to the radial distribution file
303     if (fp_rdist != NULL && i >= its_eq)
304     {
305         double bin_width = 0.05; // Width of the bins
306         double L = a_0 * 4; // Length of the supercell
307         int N_bins = (int)(L / 2 / bin_width); // Number of bins
308         double *bins = calloc(N_bins, sizeof(double)); // Bins for the radial
    ↪ distribution
309         // Write the header
310         if (i == its_eq){
311             fprintf(fp_rdist, "%i, %i, %f, %f\n", its, its_eq, delta_t, a_0);
312         }
313     }

```



```

314 // Calculate the radial distribution
315 radial_dist(bins, positions, N_bins, bin_width, N, L);
316
317 // Write the radial distribution to the file
318 for (int j = 0; j < N_bins; j++)
319 {
320     if (j == N_bins - 1)
321     {
322         fprintf(fp_rdist, "%f", bins[j]);
323     }
324     else
325     {
326         fprintf(fp_rdist, "%f, ", bins[j]);
327     }
328 }
329 fprintf(fp_rdist, "\n");
330 }
331 // Calculate the structure factor and write it to the structure factor file
332 if (fp_sfact != NULL && i >= its_eq)
333 {
334     int N_max = 10; // Maximum value in each direction for n_x, n_y and n_z
335     double L = a_0 * 4; // Length of the supercell
336     int n = 2*N_max + 1; // Number of grid points in each direction
337     int n_points = n * n * n; // Total number of grid points
338     double *S_q = (double *)malloc(n_points * sizeof(double)); // Structure
339     ↪ factor
340     int n_bins = 500; // Number of bins
341     double q_max = 2 * M_PI / L * N_max * sqrt(3); // Maximum value of the
342     ↪ magnitude of q
343     double bin_width = q_max / n_bins; // Width of the bins
344
345     // Initialize the grid and keep it throughout the iterations
346     static double ****grid = NULL;
347     if (i == its_eq)
348     {
349         grid = init_grid(N_max, L);
350     }
351     // Calculate the structure factor
352     structure_factor(grid, positions, S_q, N, n);
353     // Calculate the spherical average of the structure factor
354     spherical_avg(grid, S_q, n, n_bins, bin_width, fp_sfact);
355     // Free memory
356     if (i == its - 1)
357     {
358         destroy_grid(grid, N_max);
359         free(S_q);
360     }
361 }
362 // Free memory
363 free(T);
364 free(P);
365
366 return a_0;
367 }
368 void
369 radial_dist(double *bins, double **positions, int N_bins, double bin_width, int N, double
370 ↪ L)

```

```

369 {
370     double r; // Distance between atoms
371     double V = L * L * L; // Volume of the supercell
372     double norm_factor; // Normalization factor
373     for (int i = 0; i < N; i++)
374     {
375         for (int j = 0; j < N; j++)
376         {
377             if (i != j)
378             {
379                 // Calculate the distance between the atoms
380                 r = boundary_distance_between_vectors(positions[i], positions[j], 3, L);
381                 for (int l = 0; l < N_bins; l++)
382                 {
383                     // Check if the distance is within the bin, and increment the
384                     // bin if true
385                     if (l * bin_width < r && r < (l + 1) * bin_width)
386                     {
387                         bins[l] += 1;
388                     }
389                 }
390             }
391         }
392     }
393     for (int l = 0; l < N_bins; l++)
394     {
395         bins[l] /= (N - 1); // Average over the number of atoms
396         norm_factor = (N - 1) * 4 * M_PI * (3 * pow((l + 1), 2) - 3 * (l + 1) + 1) *
397             ↪ pow(bin_width, 3) / 3 / V;
398         bins[l] /= norm_factor; // Apply the scale factor
399     }
400 }
401 double
402 boundary_distance_between_vectors(double *v1, double *v2, int dim, double box_length)
403 {
404     double r = 0.0;
405     double delta;
406     for (int d = 0; d < dim; d++)
407     {
408         delta = v1[d] - v2[d];
409         // Apply boundary conditions
410         delta -= round(delta / box_length) * box_length;
411         r += delta * delta;
412     }
413     return sqrt(r);
414 }
415 double ****
416 init_grid(int N_max, double L)
417 {
418     int N = 2*N_max + 1; // Number of grid points in each direction
419     double ****grid = (double ****)malloc(N * sizeof(double ****)); // Initialize the grid
420     for (int i = 0; i < N; i++)
421     {
422         // Allocate memory for the grid
423         grid[i] = (double ***)malloc(N * sizeof(double **));
424         for (int j = 0; j < N; j++)

```

[illegible]

```

482         S_q_cos += cos(dot_product);
483         S_q_sin += sin(dot_product);
484     }
485     // Calculate the structure factor at the given grid point
486     Sq[it] = 1.0 / N * (S_q_cos * S_q_cos + S_q_sin * S_q_sin);
487     it++; // Increment the iterator
488 }
489 }
490 }
491 }
492 void
493 spherical_avg(double ****grid, double *Sq, int n, int n_bins, double bin_width, FILE
↳ *fp_sfact)
494 {
495     double *S_avg = calloc(n_bins, sizeof(double)); // Spherical average
496     int *counts = calloc(n_bins, sizeof(double)); // Counts for each bin
497     for (int i = 0; i < n; i++)
498     {
499         for (int j = 0; j < n; j++)
500         {
501             for (int k = 0; k < n; k++)
502             {
503                 // Calculate the magnitude of the q vector
504                 double q = sqrt(grid[i][j][k][0] * grid[i][j][k][0] +
505                                grid[i][j][k][1] * grid[i][j][k][1] +
506                                grid[i][j][k][2] * grid[i][j][k][2]);
507                 int bin = (int)(q / bin_width); // Bin index
508                 // Bin index cannot exceed the number of bins
509                 if (bin == n_bins){
510                     bin = n_bins - 1;
511                 }
512                 // Calculate the spherical average and increment the count
513                 S_avg[bin] += Sq[i * n * n + j * n + k];
514                 counts[bin]++;
515             }
516         }
517     }
518     // Write the spherical average to the file if the bin is not empty
519     for (int b = 0; b < n_bins; b++) {
520         if (counts[b] > 0)
521         {
522             if (b == n_bins - 1){
523                 fprintf(fp_sfact, "%f\n", S_avg[b] / counts[b]);
524             }
525             else{
526                 fprintf(fp_sfact, "%f, ", S_avg[b] / counts[b]);
527             }
528         }
529     }
530     free(S_avg);
531     free(counts);
532 }

```

## B.2 tools.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <gsl/gsl_fft_real.h>
5  #include <gsl/gsl_fft_halfcomplex.h>
6  #include <complex.h>
7
8  #include "tools.h"
9
10 void
11 elementwise_addition(
12     double *res,
13     double *v1,
14     double *v2,
15     unsigned int len
16 )
17 {
18     for (int i = 0; i < len; i++)
19     {
20         res[i] = v1[i] + v2[i];
21     }
22 }
23
24 void
25 elementwise_multiplication(
26     double *res,
27     double *v1,
28     double *v2,
29     unsigned int len
30 )
31 {
32     for (int i = 0; i < len; i++)
33     {
34         res[i] = v1[i] * v2[i];
35     }
36 }
37
38 void
39 addition_with_constant(
40     double *res,
41     double *v,
42     double constant,
43     unsigned int len)
44 {
45     for (int i = 0; i < len; i++)
46     {
47         res[i] = v[i] + constant;
48     }
49 }
50
51 void
52 multiplication_with_constant(
53     double *res,
54     double *v,
55     double constant,
```

```

56             unsigned int len)
57 {
58     for (int i = 0; i < len; i++)
59     {
60         res[i] = v[i] * constant;
61     }
62 }
63
64 double
65 dot_product(
66     double *v1,
67     double *v2,
68     unsigned int len
69 )
70 {
71     double res = 0.;
72     for (int i = 0; i < len; i++)
73     {
74         res += v1[i] * v2[i];
75     }
76     return res;
77 }
78
79 double **
80 create_2D_array(
81     unsigned int row_size,
82     unsigned int column_size
83 )
84 {
85     // Allocate memory for row pointers
86     double **array = malloc(row_size * sizeof(double *));
87     if (array == NULL)
88     {
89         fprintf(stderr, "Memory allocation failed for row pointers.\n");
90         return NULL;
91     }
92
93     // Allocate a single contiguous block for all elements, i.e. all matrix elements
94     // get stored in a single block of memory, array[0] is the start of the block
95     // i.e. here we have array[0][index] as the way to access the elements
96     array[0] = malloc(row_size * column_size * sizeof(double));
97     if (array[0] == NULL)
98     {
99         fprintf(stderr, "Memory allocation failed for data block.\n");
100         free(array);
101         return NULL;
102     }
103
104     // Set the row pointers to the appropriate positions in the block, in order to
105     // allow for the array[i][j] syntax. Now array[i] points to the start of the
106     // i-th row.
107     for (int i = 1; i < row_size; i++)
108     {
109         array[i] = array[0] + i * column_size;
110     }
111
112     return array;

```

```

113 }
114
115 void
116 destroy_2D_array(
117     double **array
118 )
119 {
120     if (array != NULL)
121     {
122         free(array[0]); // Free the contiguous block
123         free(array); // Free the row pointers
124     }
125 }
126
127 void
128 matrix_vector_multiplication(
129     double *result,
130     double **A,
131     double *b,
132     unsigned int n,
133     unsigned int m
134 )
135 {
136     for (int i = 0; i < n; i++)
137     {
138         result[i] = 0.;
139         for (int j = 0; j < m; j++)
140         {
141             result[i] += A[i][j] * b[j];
142         }
143     }
144 }
145
146 void
147 matrix_matrix_multiplication(
148     double **result,
149     double **A,
150     double **B,
151     unsigned int n,
152     unsigned int m,
153     unsigned int k
154 )
155 {
156     for (int i = 0; i < n; i++)
157     {
158         for (int kappa = 0; kappa < k; kappa++)
159         {
160             result[i][kappa] = 0.;
161             for (int j = 0; j < m; j++)
162             {
163                 result[i][kappa] += A[i][j] * B[j][kappa];
164             }
165         }
166     }
167 }
168
169 double

```

```

170 vector_norm(
171     double *v,
172     unsigned int len
173 )
174 {
175     double res = 0.;
176     for (int i = 0; i < len; i++)
177     {
178         res += v[i] * v[i];
179     }
180     return sqrt(res);
181 }
182
183
184 void
185 normalize_vector(
186     double *v,
187     unsigned int len
188 )
189 {
190     double norm = vector_norm(v, len);
191     multiplication_with_constant(v, v, 1. / norm, len);
192 }
193
194 double
195 average(
196     double *v,
197     unsigned int len
198 )
199 {
200     double res = 0.;
201     for (int i = 0; i < len; i++)
202     {
203         res += v[i];
204     }
205     return res / len;
206 }
207
208
209 double
210 standard_deviation(
211     double *v,
212     unsigned int len
213 )
214 {
215     double avg = average(v, len);
216     double res = 0.;
217     for (int i = 0; i < len; i++)
218     {
219         res += (v[i] - avg) * (v[i] - avg);
220     }
221     return sqrt(res / len);
222 }
223
224 double
225 distance_between_vectors(
226     double *v1,

```



```

227         double *v2,
228         unsigned int len
229     )
230 {
231     double res = 0.;
232     // With previous defined functions
233     double *diff = malloc(len * sizeof(double));
234     multiplication_with_constant(v1, v1, -1., len);
235     elementwise_addition(diff, v1, v2, len);
236     res = vector_norm(diff, len);
237     free(diff);
238     return res;
239 }
240
241 void
242 cumulative_integration(
243     double *res,
244     double *v,
245     double dx,
246     unsigned int v_len
247 )
248 {
249     double sum = 0.;
250     res[0] = 0.;
251     for (int i = 1; i < v_len; i++)
252     {
253         sum = 0.5 * (v[i - 1] + v[i]) * dx;
254         res[i] = res[i - 1] + sum;
255     }
256 }
257
258 void
259 write_xyz(
260     FILE *fp,
261     char *symbol,
262     double **positions,
263     double **velocities,
264     double alat,
265     int natoms)
266 {
267     fprintf(fp, "%i\nLattice=\"%f 0.0 0.0 0.0 %f 0.0 0.0 0.0 %f\" ", natoms, alat, alat,
268 ↪ alat);
269     fprintf(fp, "Properties=species:S:1:pos:R:3:vel:R:3 pbc=\"T T T\"\n");
270     for(int i = 0; i < natoms; ++i)
271     {
272         fprintf(fp, "%s %f %f %f %f %f %f\n",
273             symbol, positions[i][0], positions[i][1], positions[i][2],
274             velocities[i][0], velocities[i][1], velocities[i][2]);
275     }
276 }
277
278 void fft_freq(
279     double *res,
280     int n,
281     double timestep)
282 {
283     for (int i = 0; i < n; i++)

```

```

283     {
284         if (i < n / 2)
285         {
286             res[i] = 2 * M_PI * i / (n * timestep);
287         }
288         else
289         {
290             res[i] = 2 * M_PI * (i - n) / (n * timestep);
291         }
292     }
293 }
294
295 /* Freely given functions */
296 void
297 skip_line(FILE *fp)
298 {
299     int c;
300     while (c = fgetc(fp), c != '\n' && c != EOF);
301 }
302
303 void
304 read_xyz(
305     FILE *fp,
306     char *symbol,
307     double **positions,
308     double **velocities,
309     double *alat)
310 {
311     int natoms;
312     if(fscanf(fp, "%i\nLattice=\"%lf 0.0 0.0 0.0 %lf 0.0 0.0 0.0 %lf\" ", &natoms, alat,
313 ↪ alat, alat) == 0){
314         perror("Error");
315     }
316     skip_line(fp);
317     for(int i = 0; i < natoms; ++i)
318     {
319         fscanf(fp, "%s %lf %lf %lf ",
320             symbol, &positions[i][0], &positions[i][1], &positions[i][2]);
321         fscanf(fp, "%lf %lf %lf\n",
322             &velocities[i][0], &velocities[i][1], &velocities[i][2]);
323     }
324 }
325
326 void powerspectrum(
327     double *res,
328     double *signal,
329     int n,
330     double timestep)
331 {
332     /* Declaration of variables */
333     double *complex_coefficient = malloc(sizeof(double) * 2*n); // array for the complex
334     ↪ fft data
335     double *data_cp = malloc(sizeof(double) * n);
336
337     /*make copy of data to avoid messing with data in the transform*/
338     for (int i = 0; i < n; i++)
339     {

```

```

338     data_cp[i] = signal[i];
339 }
340
341 /* Declare wavetable and workspace for fft */
342 gsl_fft_real_wavetable *real;
343 gsl_fft_real_workspace *work;
344
345 /* Allocate space for wavetable and workspace for fft */
346 work = gsl_fft_real_workspace_alloc(n);
347 real = gsl_fft_real_wavetable_alloc(n);
348
349 /* Do the fft*/
350 gsl_fft_real_transform(data_cp, 1, n, real, work);
351
352 /* Unpack the output into array with alternating real and imaginary part */
353 gsl_fft_halfcomplex_unpack(data_cp, complex_coefficient, 1, n);
354
355 /*fill the output powspec_data with the powerspectrum */
356 for (int i = 0; i < n; i++)
357 {
358     res[i] = (complex_coefficient[2*i]*complex_coefficient[2*i]+
359             complex_coefficient[2*i+1]*complex_coefficient[2*i+1]);
360     res[i] *= timestep / n;
361 }
362
363 /* Free memory of wavetable and workspace */
364 gsl_fft_real_wavetable_free(real);
365 gsl_fft_real_workspace_free(work);
366 free(complex_coefficient);
367 free(data_cp);
368 }

```

## C Source code in Python

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  import scipy.constants as K
4  import pandas as pd
5
6  # Latex style
7  plt.style.use('default')
8  plt.rc('text', usetex=True)
9  plt.rc('font', family='serif')
10 plt.rc('font', size=20)
11 plt.rcParams['text.latex.preamble'] = r'\usepackage{amsmath}'
12
13 # Set ticks on both sides
14 plt.rcParams['xtick.direction'] = 'in'
15 plt.rcParams['ytick.direction'] = 'in'
16 plt.rcParams['xtick.major.size'] = 5
17 plt.rcParams['ytick.major.size'] = 5
18 plt.rcParams['xtick.top'] = True
19 plt.rcParams['ytick.right'] = True
20

```

```

21 # Constants
22 k_B = K.Boltzmann
23 e = K.elementary_charge
24 k_B /= e
25 n_atoms = 256
26
27 # Functions
28 def read_data(delta_t, its, its_eq, task, opt=None):
29     if task == 2:
30         filename = f'data/task_2/data_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
31     elif task == 3:
32         filename = f'data/task_3/data_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
33     elif task == 4:
34         filename = f'data/task_4/data_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
35     elif task == 6:
36         filename = f'data/task_4/rdist_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
37     elif task == 7:
38         if opt is not None:
39             filename =
39                 ↪ f'data/task_4/sfact_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}_{opt}.csv'
40         else:
41             filename = f'data/task_4/sfact_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
42     if task not in [6, 7]:
43         data_facts = np.genfromtxt(filename, dtype=np.float64, delimiter=',', max_rows=2)
44         data = np.genfromtxt(filename, dtype=np.float64, encoding=None,
44             ↪ delimiter=',')[3:, :]
45         its, t_max, delta_t, its_eq, _, _, _ = data_facts[1]
46         a_0, N_bins, N_max = None, None, None
47     elif task == 6:
48         data_facts = np.genfromtxt(filename, dtype=np.float64, delimiter=',', max_rows=2)
49         data = np.genfromtxt(filename, dtype=np.float64, encoding=None, delimiter=',',
49             ↪ skip_header=2)
50         N_bins = data.shape[1]
51         its, its_eq, delta_t, a_0 = data_facts[1]
52         t_max, N_max = None, None
53     elif task == 7:
54         data_facts = np.genfromtxt(filename, dtype=np.float64, delimiter=',', max_rows=2)
55         data = np.genfromtxt(filename, dtype=np.float64, encoding=None, delimiter=',',
55             ↪ skip_header=2)
56         N_bins = data.shape[1] - 1
57         its, its_eq, delta_t, a_0, N_max, _ = data_facts[1]
58         t_max = None
59
60     return data, int(its), t_max, delta_t, int(its_eq), N_bins, a_0, N_max
61
62 def plot_trajs(its_eq, its, t_max, delta_t, task, save=True):
63
64     def read_trajs():
65         if task == 3:
66             filename = f'data/task_3/trajs_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
67         elif task == 4:
68             filename = f'data/task_4/trajs_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
69         trajs = np.genfromtxt(filename, dtype=np.float64, encoding=None, delimiter=',')
70
71         return trajs[3:, :]
72
73

```

```

74     trajs = read_trajs()
75
76     fig, axs = plt.subplots(2, 2, figsize=(12, 7), sharex=True)
77
78     its_eq = 0
79     t = np.linspace(0, t_max - t_max*its_eq/its, int(its-its_eq))
80
81     axs[0][0].plot(t, trajs[its_eq:, 0], label='$x_1$')
82     axs[0][0].plot(t, trajs[its_eq:, 1], label='$y_1$')
83     axs[0][0].plot(t, trajs[its_eq:, 2], label='$z_1$')
84     axs[0][0].set_title('Atom 1')
85
86     axs[0][1].plot(t, trajs[its_eq:, 3], label='$x_2$')
87     axs[0][1].plot(t, trajs[its_eq:, 4], label='$y_2$')
88     axs[0][1].plot(t, trajs[its_eq:, 5], label='$z_2$')
89     axs[0][1].set_title('Atom 2')
90
91     axs[1][0].plot(t, trajs[its_eq:, 6], label='$x_3$')
92     axs[1][0].plot(t, trajs[its_eq:, 7], label='$y_3$')
93     axs[1][0].plot(t, trajs[its_eq:, 8], label='$z_3$')
94     axs[1][0].set_title('Atom 3')
95
96     axs[1][1].plot(t, trajs[its_eq:, 9], label='$x_4$')
97     axs[1][1].plot(t, trajs[its_eq:, 10], label='$y_4$')
98     axs[1][1].plot(t, trajs[its_eq:, 11], label='$z_4$')
99     axs[1][1].set_title('Atom 4')
100
101     for ax in axs.flatten():
102         ax.legend(loc='upper right', ncol=3)
103         y_min, y_max = ax.get_ylim()
104         ax.set_ylim(y_min, y_max + 0.5 * (y_max - y_min))
105
106     axs[0][0].set_ylabel('Position (Å)')
107     axs[1][0].set_ylabel('Position (Å)')
108     axs[1][0].set_xlabel('Time (ps)')
109     axs[1][1].set_xlabel('Time (ps)')
110
111     plt.tight_layout()
112
113     return fig
114
115 def plot_T_E(data, its, its_eq, t_max, delta_t, save=True):
116     def init_TE_fig():
117         fig, axs = plt.subplots(2, 2, figsize=(12, 7), sharex=True)
118         fig.subplots_adjust(hspace=0.05) # adjust space between Axes
119         ax1, ax2, ax3, ax4 = axs[0][0], axs[1][0], axs[0][1], axs[1][1]
120         ax3.set_visible(False)
121         ax4.set_visible(False)
122
123         ax1_2 = fig.add_subplot(1, 2, 1, frameon=False)
124         ax1_2.set_ylabel('Energy [eV]', labelpad=50)
125         ax1_2.set_xticks([])
126         ax1_2.set_yticks([])
127
128         ax3 = fig.add_subplot(1, 2, 2)
129         ax3.set_ylabel('Temperature [°C]', labelpad=10)
130         plt.suptitle(f'\\textbf{{Energies and Temperature vs. time $\\vert$\\vert$\\hspace{{0.3cm}}\\Delta t = {delta_t}$ ps}}')

```

```

131     d = .5 # proportion of vertical to horizontal extent of the slanted line
132     kwargs = dict(marker=[(-1, -d), (1, d)], markersize=12,
133                     linestyle="none", color='k', mec='k', mew=1, clip_on=False)
134     ax1.plot([0, 1], [0, 0], transform=ax1.transAxes, **kwargs)
135     ax2.plot([0, 1], [1, 1], transform=ax2.transAxes, **kwargs)
136
137     ax1.spines.bottom.set_visible(False)
138     ax2.spines.top.set_visible(False)
139
140
141     return fig, ax1, ax2, ax3, ax4
142
143 t = np.linspace(0, t_max, int(its))
144 E_k_min, E_k_max = np.min(data[1:, 0]), np.max(data[1:, 0])
145 E_p_min, E_p_max = np.min(data[1:, 1]), np.max(data[1:, 1])
146
147 fig, ax1, ax2, ax3, ax4 = init_TE_fig()
148 ax1.plot(t, data[:, 0], label='Kinetic Energy', linewidth=3)
149 ax1.plot(t, data[:, 1], label='Potential Energy', linewidth=3)
150 ax1.plot(t, data[:, 2], label='Total Energy', linewidth=3)
151 ax2.plot(t, data[:, 0], label='Kinetic Energy', linewidth=3)
152 ax2.plot(t, data[:, 1], label='Potential Energy', linewidth=3)
153 ax2.plot(t, data[:, 2], label='Total Energy', linewidth=3)
154 ax1.set_ylim(E_k_max - 0.75 * (E_k_max - E_k_min), E_k_max + 0.5 * (E_k_max -
155     ↪ E_k_min))
156 ax2.set_ylim(E_p_min - 0.1 * (E_p_max - E_p_min), E_p_max + 0.25 * (E_p_max -
157     ↪ E_p_min))
158
159 ax3.plot(t, data[:, 3] - 273.15, label='Temperature', color='tab:red', linewidth=3)
160 for ax in (ax2, ax3):
161     ax.set_xlabel('Time [ps]')
162
163 ax1.tick_params(axis='x', which='both', bottom=False, top=True, labelbottom=False)
164 ax2.tick_params(axis='x', which='both', bottom=True, top=False, labelbottom=True)
165 ax1.legend(fontsize=18, loc='upper right')
166 ax3.legend(fontsize=18, loc='upper right')
167 plt.tight_layout()
168
169 return fig
170
171 def plot_T_P(data, its_eq, save=False):
172     def init_TP_fig():
173         fig, ax1 = plt.subplots(1, 1, figsize=(8, 6))
174         ax2 = ax1.twinx()
175         ax1.set_ylabel('Temperature [°C]', labelpad=10, color='tab:red')
176         ax2.set_ylabel('Pressure [MPa]', labelpad=10, color='tab:blue')
177
178     for ax in (ax1, ax2):
179         if ax == ax1:
180             ax.tick_params(axis='y', direction='in', length=5, width=1,
181                 ↪ colors='tab:red', right=False, pad=10)
182         else:
183             ax.tick_params(axis='y', direction='in', length=5, width=1,
184                 ↪ colors='tab:blue', left=False, pad=10)
185
186     return fig, ax1, ax2

```

```

184     T_avg = data[:, 4] - 273.15 # Convert to Celsius
185     P_avg = data[:, 6]
186     t = np.linspace(0, t_max, its)
187
188     fig, ax1, ax2 = init_TP_fig()
189     ax1.plot(t, T_avg, color='tab:red', label='Temperature')
190     ax2.plot(t, P_avg, color='tab:blue', label='Pressure')
191     ax1.set_xlabel('Time [ps]')
192     ax1.set_yticks([0, 500, 1000])
193     ax2.set_ylim(-500, 1000)
194     ax2.set_yticks([-500, 0, 500, 1000])
195     ax1.legend(loc='upper left', labelcolor='tab:red')
196     ax2.legend(loc='upper right', labelcolor='tab:blue')
197     plt.tight_layout()
198
199     return fig
200
201 def plot_volume_evo(data, its, t_max, delta_t, its_eq, save=True):
202     volume_unit_cell = np.power(data[:, 7], 3)
203     t = np.linspace(0, t_max, its)
204
205     fig, ax = plt.subplots(figsize=(8, 6))
206     plt.plot(t, volume_unit_cell)
207     plt.xlabel('Time [ps]')
208     plt.ylabel('Unit Cell Volume [Å3]')
209     plt.tight_layout()
210
211     return fig
212
213 def save_fig(fig, name, task):
214     fig.savefig(f'figs/task_{task}/{name}.pdf')
215
216 ##### Task 1 #####
217 save = False
218 data = np.loadtxt('data/task_1/energies.csv', delimiter=',')
219
220 fig, ax = plt.subplots()
221
222 plt.plot(np.power(data[:, 0], 3), data[:, 1] / 64, 'x', ms=10, label='Data Points')
223 fit = np.polyfit(np.power(data[:, 0], 3), data[:, 1] / 64, 2)
224 plt.plot(np.power(data[:, 0], 3), np.polyval(fit, np.power(data[:, 0], 3)), 'k--',
225          label='Quadratic Fit')
226
227 print(f'The unit cell volume generating the minimum energy is {np.power(-fit[1] / (2 *
228          ↪ fit[0]), 1):.3f} Å3')
229 print(f'The lattice parameter value generating the minimum energy is {np.power(-fit[1] /
230          ↪ (2 * fit[0]), 1/3):.3f} Å')
231
232 plt.xlabel('Unit Cell Volume (Å3)')
233 plt.ylabel('Energy/unit cell (eV)')
234 plt.legend()
235 plt.tight_layout()
236 plt.show()
237
238 if save:
239     save_fig(fig, 'energy_vs_volume', 1)

```

```

238 ##### Task 2 #####
239 t_max = 10
240
241 save = False
242 data_01, its_01, t_max_01, delta_t_01, _, _ , _ , _ = read_data(0.1, t_max/0.1, 0, 2)
243 data_002, its_002, t_max_002, delta_t_002, _, _ , _ , _ = read_data(0.02, t_max/0.02, 0,
↳ 2)
244 data_0001, its_0001, t_max_0001, delta_t_0001, _, _ , _ , _ = read_data(0.001,
↳ t_max/0.001, 0, 2)
245
246 fig_01 = plot_T_E(data_01, its_01, None, t_max_01, delta_t_01)
247 fig_002 = plot_T_E(data_002, its_002, None, t_max_002, delta_t_002)
248 fig_0001 = plot_T_E(data_0001, its_0001, None, t_max_0001, delta_t_0001)
249
250 if save:
251     save_fig(fig_01, 'TE_plot_dt01', 2)
252     save_fig(fig_005, 'TE_plot_dt005', 2)
253     save_fig(fig_0001, 'TE_plot_dt0001', 2)
254
255 ##### Task 3 #####
256 save = False
257 data, its, t_max, delta_t, its_eq, _, _ , _ = read_data(0.001, 40000, 25000, task=3)
258
259 fig_TE = plot_T_E(data, its, its_eq, t_max, delta_t)
260 fig_TP = plot_T_P(data, its_eq)
261 fig_evo = plot_volume_evo(data, its, t_max, delta_t, its_eq)
262 fig_trajs = plot_trajs(its_eq, its, t_max, delta_t, task=3)
263
264 if save:
265     save_fig(fig_TE, 'TE_plot', 3)
266     save_fig(fig_TP, 'TP_plot', 3)
267     save_fig(fig_evo, 'volume_evo', 3)
268     save_fig(fig_trajs, 'trajs_3', 3)
269
270 ##### Task 4 #####
271 save = False
272 task = 4
273 data, its, t_max, delta_t, its_eq, _, _ , _ = read_data(0.001, 50000, 30000, task)
274
275 fig_trajs = plot_trajs(its_eq, its, t_max, delta_t, task)
276 fig_evo = plot_volume_evo(data, its, t_max, delta_t, its_eq)
277 fig_TP = plot_T_P(data, its_eq)
278 fig_TE = plot_T_E(data, its, its_eq, t_max, delta_t)
279
280 if save:
281     save_fig(fig, 'trajs_4', task)
282     save_fig(fig_evo, 'volume_evo', task)
283     save_fig(fig_TP, 'TP_plot', task)
284     save_fig(fig_TE, 'TE_plot', task)
285
286 data, its, t_max, delta_t, its_eq, _, _ , _ = read_data(0.001, 50000, 50000, task)
287
288 fig_trajs = plot_trajs(its_eq, its, t_max, delta_t, task)
289 fig_evo = plot_volume_evo(data, its, t_max, delta_t, its_eq)
290 fig_TP = plot_T_P(data, its_eq)
291 fig_TE = plot_T_E(data, its, its_eq, t_max, delta_t)
292

```



```

293 if save:
294     save_fig(fig_trajs, 'warm_up/trajs_4_warmup', task)
295     save_fig(fig_evo, 'warm_up/volume_evo_warmup', task)
296     save_fig(fig_TP, 'warm_up/TP_plot_warmup', task)
297     save_fig(fig_TE, 'warm_up/TE_plot_warmup', task)
298
299 ##### Task 5 #####
300 def avg_fluct_square(data):
301     data_avg = np.mean(data)
302     d_data = np.mean((data - data_avg)**2)
303
304     return d_data
305
306 def CV(data, its_eq, n_atoms, k_B, e):
307     T_avg = np.mean(data[its_eq:, 4])
308     dE_kin = avg_fluct_square(data[its_eq:, 0])
309     dE_pot = avg_fluct_square(data[its_eq:, 1])
310
311     Cv_kin = (3 * n_atoms * k_B / 2) / (1 - 2 / (3 * n_atoms * k_B**2 * T_avg**2)) *
312     ↪ dE_kin) * e
313     Cv_pot = (3 * n_atoms * k_B / 2) / (1 - 2 / (3 * n_atoms * k_B**2 * T_avg**2)) *
314     ↪ dE_pot) * e
315
316     return Cv_kin, Cv_pot, T_avg, dE_kin, dE_pot
317
318 data_3, _, _, _, its_eq_3, _, _, _ = read_data(0.001, 40000, 25000, task=3)
319 data_4, _, _, _, its_eq_4, _, _, _ = read_data(0.001, 50000, 30000, task=4)
320
321 Cv_kin_3, Cv_pot_3, T_avg_3, dE_kin_3, dE_pot_3 = CV(data_3, its_eq_3, n_atoms, k_B, e)
322 Cv_kin_4, Cv_pot_4, T_avg_4, dE_kin_4, dE_pot_4 = CV(data_4, its_eq_4, n_atoms, k_B, e)
323
324 results = pd.DataFrame({'Cv_kin': [Cv_kin_3, Cv_kin_4],
325                         'Cv_pot': [Cv_pot_3, Cv_pot_4],
326                         'dE_kin': [dE_kin_3, dE_kin_4],
327                         'dE_pot': [dE_pot_3, dE_pot_4],
328                         'T_avg': [T_avg_3, T_avg_4]},
329                         index=['Solid', 'Liquid'], )
330 results = results.style.format({"Cv_kin": "{:.3e}", "Cv_pot": "{:.3e}",
331                                "dE_kin": "{:.3f}", "dE_pot": "{:.3f}", "T_avg": "{:.3f}"})
332 display(results)
333
334 ##### Task 6 #####
335 def plot_rdf(r, rdist, r_m, r_m_ind, n, save=False):
336     fig, ax = plt.subplots(figsize=(8, 6))
337     plt.plot(r, rdist, 'k', label='RDF', lw=2.5)
338     plt.axvline(r_m, ymin=0, ymax=0.2, color='k', linestyle='--')
339     plt.plot(r[r_m_ind], rdist[r_m_ind], '*', color='tab:orange', markersize=15,
340     ↪ label='$r_m$')
341     plt.fill_between(r[:r_m_ind+1], rdist[:r_m_ind+1], alpha=0.5, color='k')
342     plt.ylim(0, 3)
343     plt.xlabel('$r$ [Å]')
344     plt.ylabel('$g(r)$')
345     plt.legend(loc='upper right')
346     plt.tight_layout()
347
348     return fig

```

```

347 save = False
348 rdist_data, its, t_max, delta_t, its_eq, N_bins, a_0, _ = read_data(0.001, 50000, 30000,
↪ 6)
349 rdist = np.cumsum(rdist_data, axis=0)[-1, :]/len(rdist_data)
350
351 r = np.linspace(0, a_0*2, N_bins)
352 r_m_ind = np.argmin(rdist[50:]) + 50
353 r_m = r[r_m_ind]
354 n = n_atoms / np.power(4*a_0, 3)
355
356 I_r = 4 * n * np.pi * np.trapezoid(rdist[:r_m_ind] * r[:r_m_ind]**2, r[:r_m_ind])
357 print(f'The coordination number is {I_r:.3f}')
358
359 fig = plot_rdf(r, rdist, r_m, r_m_ind, n)
360
361 if save:
362     save_fig(fig, 'rdf', 6)
363
364 ##### Task 7 #####
365 def plot_sfact(q, sfact):
366     fig, ax = plt.subplots(figsize=(8, 6))
367
368     plt.plot(q, sfact, 'k', label='Structure Factor', lw=2.5)
369
370     plt.xlabel('$q$ [Å-1 $]$')
371     plt.ylabel('$S(q)$')
372     plt.legend(loc='upper right')
373     plt.tight_layout()
374
375     return fig
376
377 save = False
378 sfact_data, its, _, delta_t, its_eq, N_bins, a_0, N_max = read_data(0.001, 50000, 30000,
↪ 7, opt='big')
379 sfact = np.cumsum(sfact_data, axis=0)[-1, 1:]/len(sfact_data)
380
381 q_max = 2 * np.pi / (4*4.25) * N_max * np.sqrt(3)
382 q = np.linspace(0, q_max, int(N_bins))
383 print(f'The first peak of the structure factor is at {q[np.argmax(sfact)]:.3f}
↪ Å-1 $')
384
385 fig = plot_sfact(q, sfact)
386
387 if save:
388     save_fig(fig, 'sfact', 7)

```