Homework 1:
Simulation of Molecular Dynamics

Jonatan Haraldsson
jonhara@chalmers.se

Oscar Stommendal
oscarsto@chalmers.se

Task №	Points	Avail. points
$\sum$		

December 6th 2024
Course: *Computational Physics, 7.5 hp*
Course Code: *FKA 122*

Physics, MSc.
Chalmers University of Technology

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.

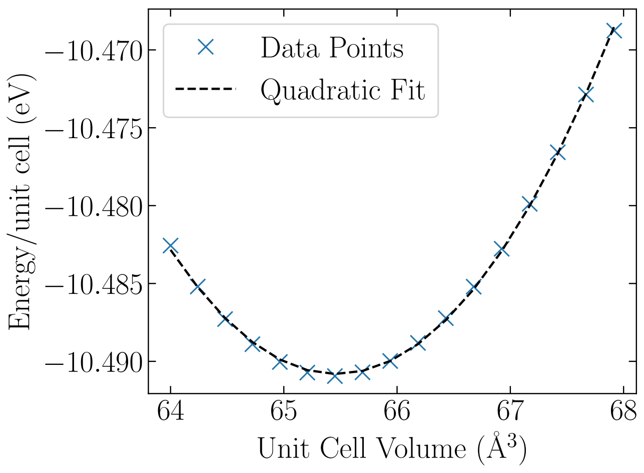

Figure 1: Potential energy per unit cell for initialized FCC structures with different lattice parameter a_0 (note that we have $V = a_0^3$ on the x-axis), with a quadratic fit.

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 Δt. 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 $a_0 = 4.03\,\text{Å}$ found in Task 1 using `gsl_rng` (random generator)[*].

Code Snippet Task 2, Verlet

```
// Calculate the initial forces, potential energy and virial
calculate(&E_pot, &virial, forces, positions, a_0 * cell_length, N);
for (unsigned int i = 0; i < its; i++)
    {
    E_kin = 0.0;
    // Perform the first half step
    for (unsigned int j = 0; j < N; j++)
    {
        for (unsigned int k = 0; k < 3; k++)
        {
            velocities[j][k] += 0.5 * delta_t * forces[j][k] / m;
            positions[j][k] += delta_t * velocities[j][k];
        }
    }
    // Calculate new accelerations
    calculate(&E_pot, &virial, forces, positions, a_0 * cell_length, N);
    // Perform the second half step
    for (unsigned int j = 0; j < N; j++)
    {
        for (unsigned int k = 0; k < 3; k++)
        {
            velocities[j][k] += 0.5 * delta_t * forces[j][k] / m;
        }
        E_kin += 0.5 * m * vector_norm(velocities[j], 3) *
        ↪ vector_norm(velocities[j], 3);
    }
}
```

To find a fitting time-step (Δt) for the simulation, the system's energy, i.e. potential and kinetic, and temperature were plotted for different Δt. 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 $\sim 620\,°\text{C}$ (900 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.

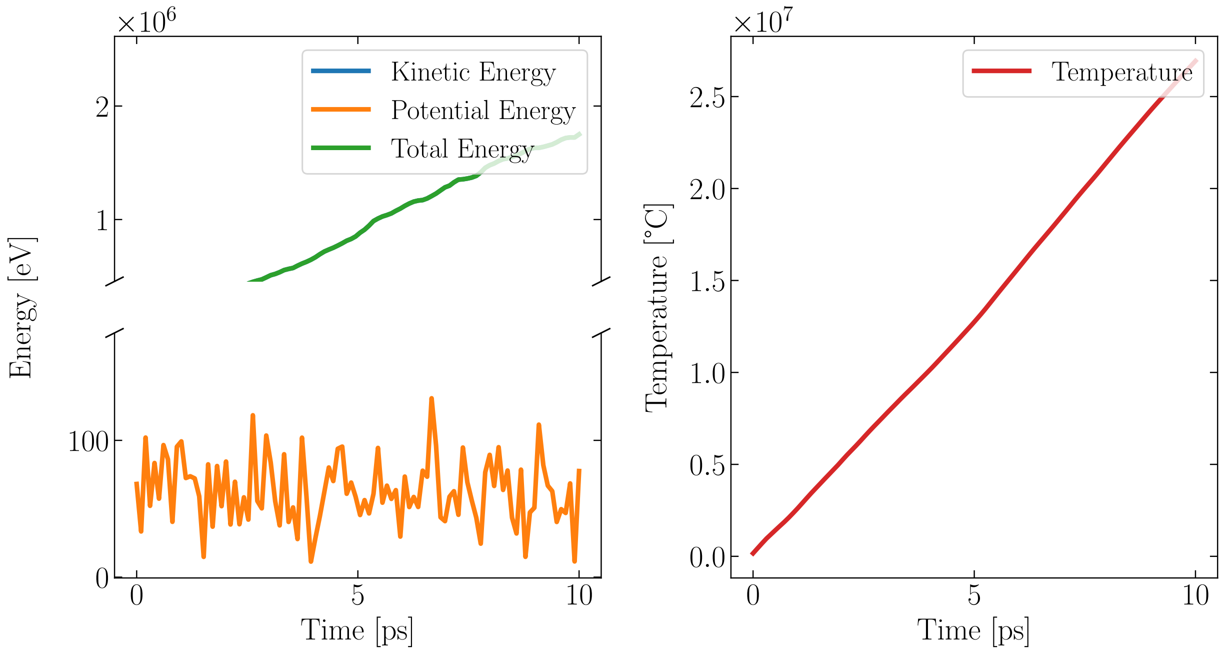

Figure 2: The plot to the left contain instantaneous kinetic, potential and total energy for the system at each time-step. The right plot displays the average temperature throughout the simulation. In this case, $\Delta t = 0.1\,\mathrm{ps}$.

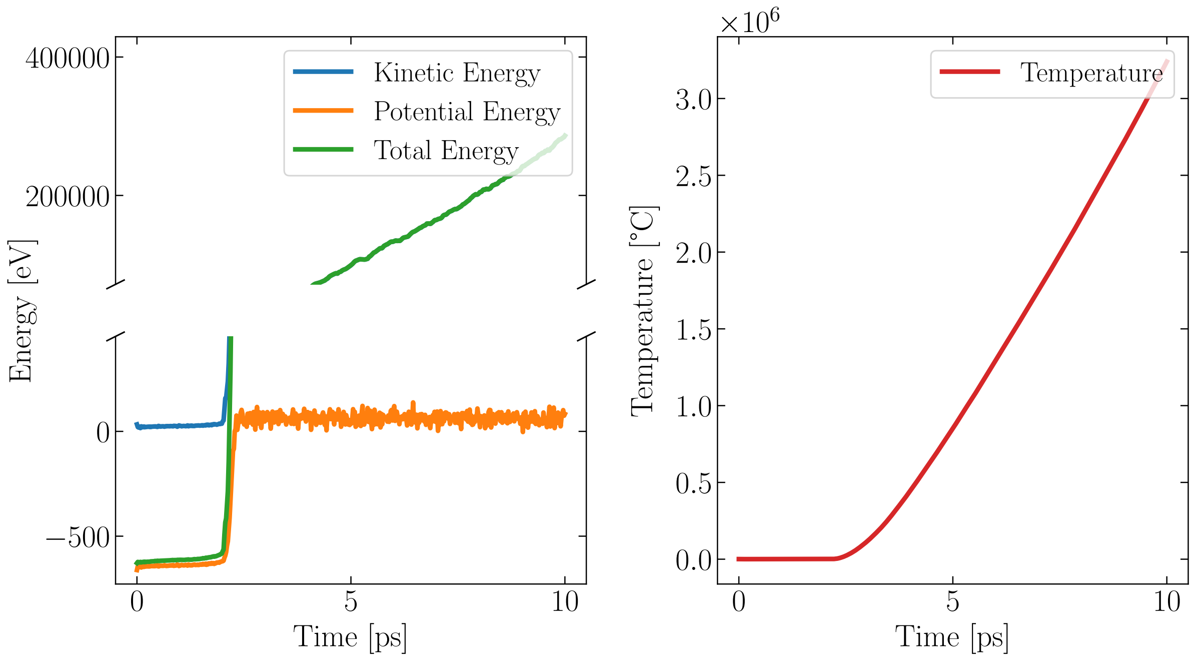

Figure 3: The plot to the left contain instantaneous kinetic, potential and total energy for the system at each time-step. The right plot displays the average temperature throughout the simulation. In this case, $\Delta t = 0.02\,\mathrm{ps}$.

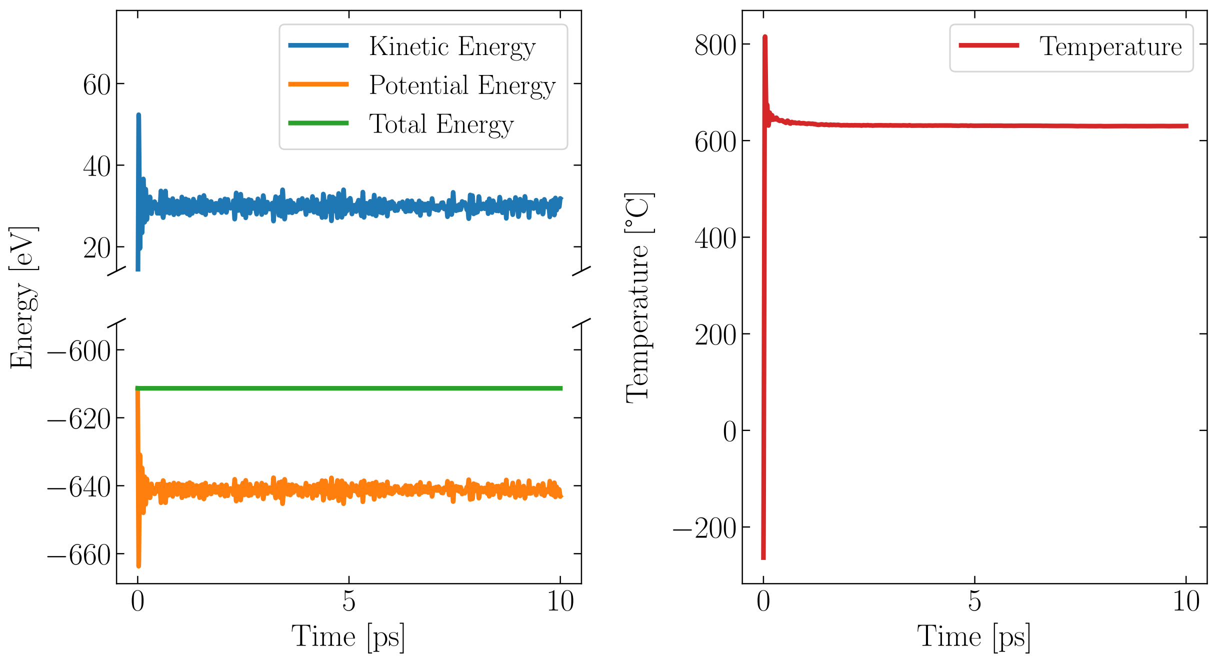

Figure 4: The plot to the left contain instantaneous kinetic, potential and total energy for the system at each time-step. The right plot displays the average temperature throughout the simulation. In this case, $\Delta t = 0.001\,\mathrm{ps}$.

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 τ_T or τ_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 $\boldsymbol{v}$ and position $\boldsymbol{r}$ for each particle was scaled accordingly

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

at the end of each iteration i in the *Verlet* function. In addition, the scaling parameters α_T and α_P are given by

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

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

For the solid state, the system was equilibrated to $T_{eq} = 500\,^\circ\mathrm{C}$ and $P_{eq} = 1\,\mathrm{bar} = 0.1\,\mathrm{MPa}$ with 25 000 iterations, during $25\,000 \cdot \Delta t = 25\,\mathrm{ps}$. The time constants τ_T and τ_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 $t_{eq} = 25\,\mathrm{ps}$, and that the temperature and pressure are converging to the set values $500\,^\circ\mathrm{C}$ and $0.1\,\mathrm{MPa}$.

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.

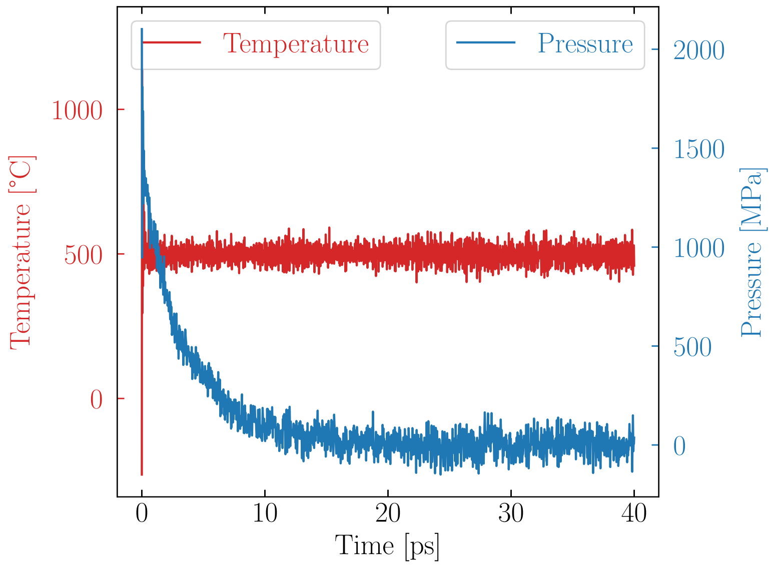

Figure 5: Equilibration of temperature and pressure with equilibration time $t_{eq} = 25$ ps, $T_{eq} = 500$ °C and $P_{eq} = 0.1$ MPa.

Is 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 ~ 70 Å^3 after roughly 15 ps. After the equilibration, the lattice parameter is hence ~ 4.12 Å, which is slightly higher than $a_0 = 4.03$ Å, 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.

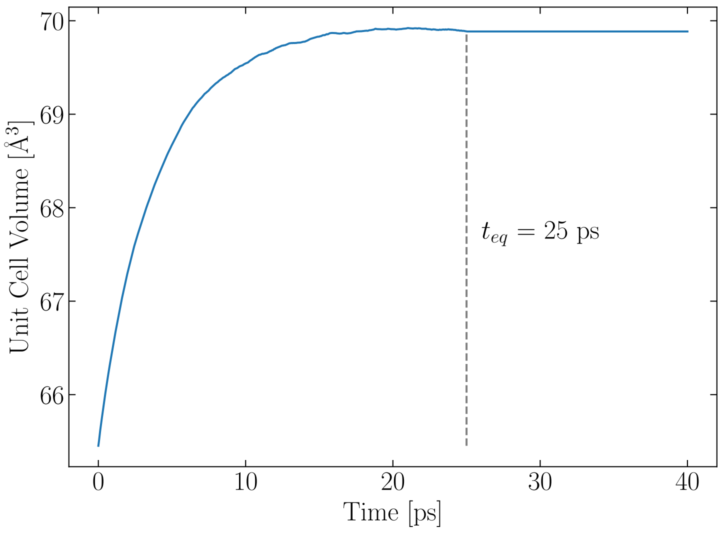

Figure 6: Unit cell volume during the equilibration ($t < t_{eq} = 25$ ps) and production ($t > t_{eq}$) simulation runs.

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.

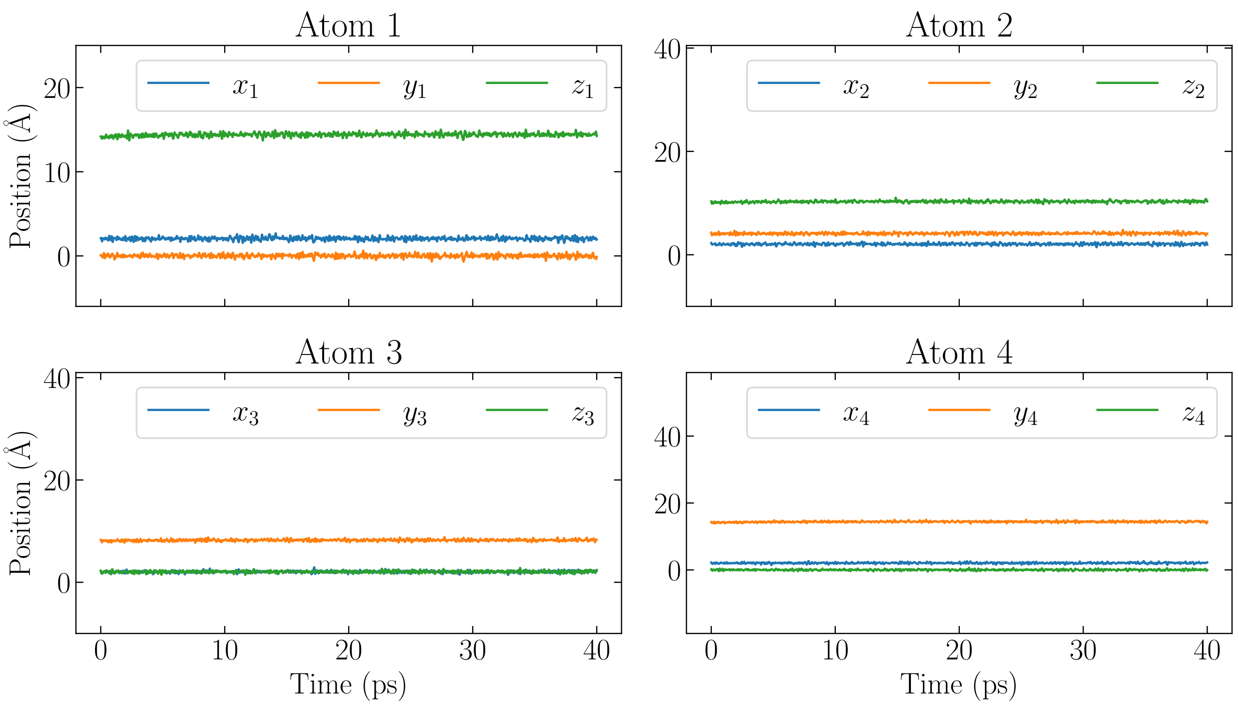

Figure 7: Atomic trajectories: x-, y- and z coordinates, for four atoms as a function of time during the equilibration (25 ps) + production (15 ps) simulation.

Task 4 – Characteristics of the Melted System

Similar to the previous task, an equilibration was made, however, this time in a liquid state at $T_{eq} = 700\,°\mathrm{C}$ and $P_{eq} = 0.1\,\mathrm{MPa}$. First, the system had to be liquefied, and that was done with an equilibration at $T = 900\,°\mathrm{C}$ (substantially higher than the melting point of aluminum $T_m \approx 660\,°\mathrm{C}$ [6]) for 50 ps. Secondly, another equilibration at $T_{eq} = 700\,°\mathrm{C}$ 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 $T_{avg} = 699.6\,°\mathrm{C}$ and average pressure of $P_{avg} = -7.7\,\mathrm{MPa}$ was measured. The temperature value deviates roughly $0.4\,°\mathrm{C}$ 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 $P_{eq} = 0.1\,\mathrm{MPa}$ is set quite low. Moreover, the pressure values in Figure 8 shows larger fluctuations $\sim \pm 100\,\mathrm{MPa}$ 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.

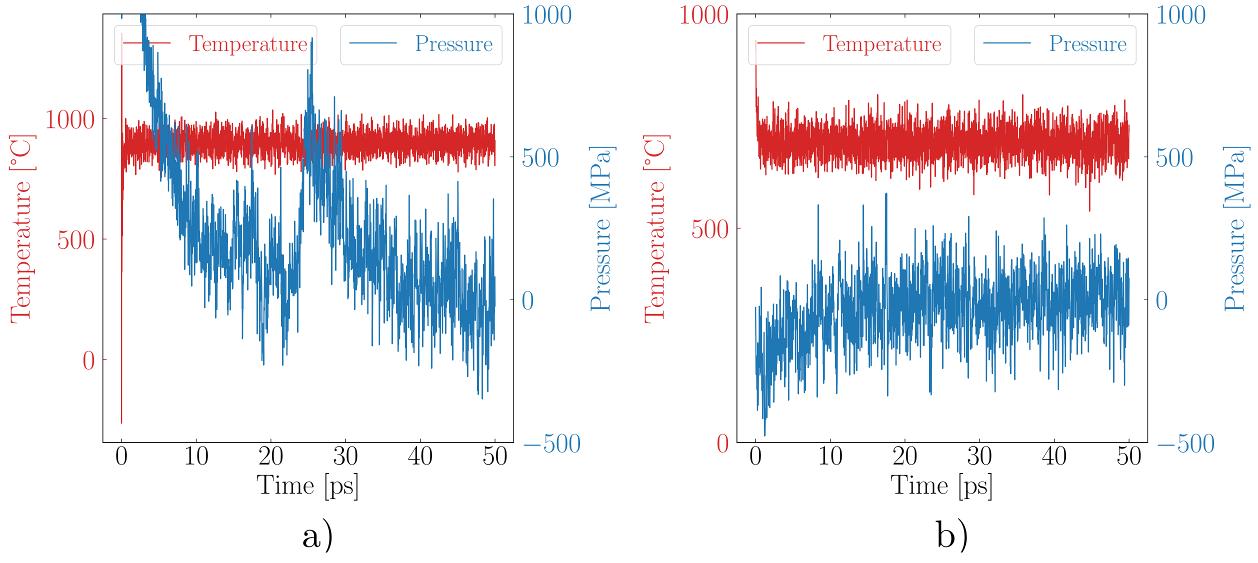

Figure 8: Temperature and pressure during the first long equilibration at $T = 900\,°C$ (a) and during the second equilibration (30 ps) at $T = 700\,°C$ and production (20 ps) simulation (b).

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.

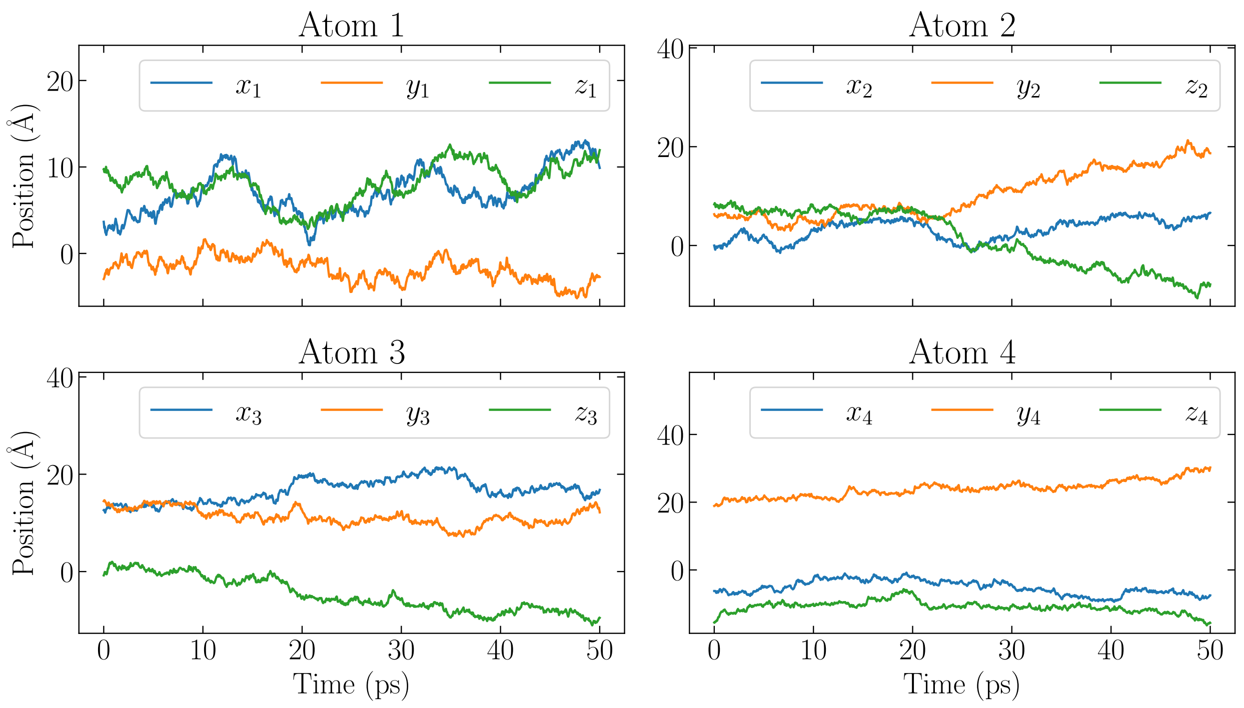

Figure 9: Atomic trajectories: x-, y- and z coordinates, for four atoms as a function of time during the second equilibration (30 ps) + production (20 ps) simulation.

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

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

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_{kin} and E_{pot} along with average temperatures (T_{avg}) for the solid and the liquid states.

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

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

$$C_V\left(\frac{N}{N_A}\right)^{-1} = \begin{cases} 1.2\times10^{-20}\left(\frac{256}{6.022\times10^{23}}\right)^{-1} \approx 28.2\,\text{J/(mol}\cdot\text{K)}, \text{(solid)} \\ 9.8\times10^{-21}\left(\frac{256}{6.022\times10^{23}}\right)^{-1} \approx 23.1\,\text{J/(mol}\cdot\text{K)}, \text{(liquid)}. \end{cases} \tag{2}$$

Comparing to tabulated values, the heat capacity for solid aluminum is roughly 30.0 J/(mol · 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 J/(mol · 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 `bdary_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

```c
void
radial_dist(double *bins, double **positions, int N_bins, double bin_width, int N,
↪   double L)
{
    double r; // Distance between atoms
    double V = L * L * L; // Volume of the supercell
    double norm_factor; // Normalization factor
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if (i != j)
            {
                // Calculate the distance between the atoms
                r = bdry_dist_between_vectors(positions[i], positions[j], 3, L);
                for (int l = 0; l < N_bins; l++)
                {
                    // Check if the distance is within the bin, and increment the
                    // bin if true
                    if (l * bin_width < r && r < (l + 1) * bin_width)
                    {
                        bins[l] += 1;
                    }
                }
            }
        }
    }
    for (int l = 0; l < N_bins; l++)
    {
        bins[l] /= (N - 1); // Average over the number of atoms
        norm_factor = (N - 1) * 4 * M_PI * (3 * pow((l + 1), 2) - 3 * l + 1) *
        ↪   pow(bin_width, 3) / 3 / V;
        bins[l] /= norm_factor; // Apply the scale factor
    }
}
```

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

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

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 out 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.

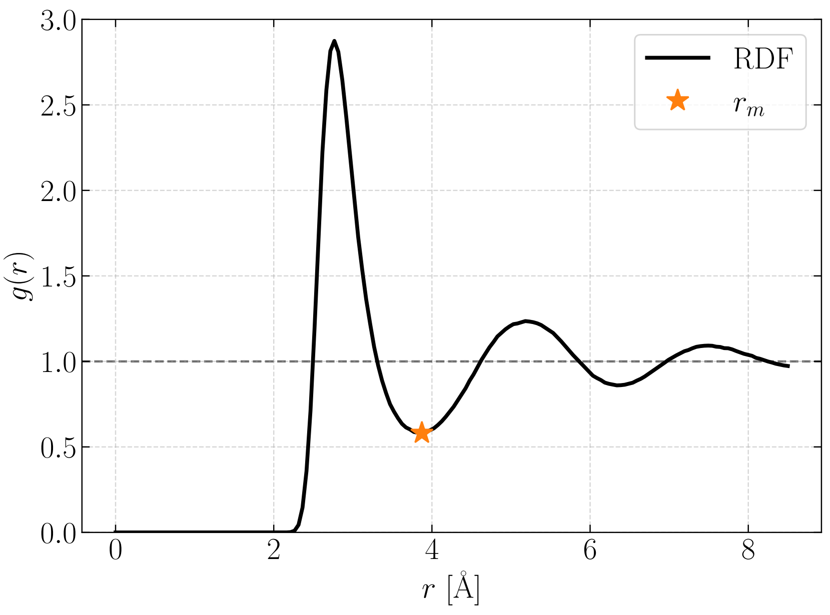

Figure 10: The radial distribution function $g(r)$ as a function of distance between the atom pairs in the lattice structure. The orange star denotes r_m, which was used to calculate the coordination number $I(r_m)$.

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

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

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, $r_m \approx 3.87$ 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), $S(\boldsymbol{q})$, for the liquid aluminum configuration. This was done using the provided formula

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

Here $\boldsymbol{r}_i(t)$ is every atom's position at time t and $\boldsymbol{q}$ is a 3-dimensional grid consistent with the periodic boundary conditions, i.e. $\boldsymbol{q} = \frac{2\pi}{L}(n_x, n_y, n_z)$, 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 $\boldsymbol{q}$-grid, a function was implemented in C, see the code snippet below.

Code snippet task 7 Initialize Grid

```c
double ****
init_grid(int N_max, double L)
{
    int N = 2*N_max + 1;
    double ****grid = (double ****)malloc(N * sizeof(double ***));
    for (int i = 0; i < N; i++)
    {
        grid[i] = (double ***)malloc(N * sizeof(double **));
        for (int j = 0; j < N; j++)
        {
            grid[i][j] = (double **)malloc(N * sizeof(double *));
            for (int k = 0; k < N; k++)
            {
                grid[i][j][k] = (double *)malloc(3 * sizeof(double));
                grid[i][j][k][0] = 2 * M_PI / L * (i - N_max);
                grid[i][j][k][1] = 2 * M_PI / L * (j - N_max);
                grid[i][j][k][2] = 2 * M_PI / L * (k - N_max);
            }
        }
    }
    return grid;
}
```

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 $\boldsymbol{q}$-vector, i.e. $q = |\boldsymbol{q}|$. The calculated values at every time-step for each $\boldsymbol{q}(q)$ was sorted into a number of bins, N_{bins} with a certain bin width Δq, see again the code snippet below.

```
Code snippet task 7, Spherical Average
```

```c
void
spherical_avg(double ****grid, double *Sq, int n, int n_bins, double bin_width)
{
    double *S_avg = calloc(n_bins, sizeof(double));
    int *counts = calloc(n_bins, sizeof(double));
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            for (int k = 0; k < n; k++)
            {
                double q = sqrt(grid[i][j][k][0] * grid[i][j][k][0] +
                                grid[i][j][k][1] * grid[i][j][k][1] +
                                grid[i][j][k][2] * grid[i][j][k][2]);
                int bin = (int)(q / bin_width);
                if (bin == n_bins){
                    bin = n_bins - 1;
                }
                S_avg[bin] += Sq[i * n * n + j * n + k];
                counts[bin]++;
            }
        }
    }
}
```

When computing the SSF, we used a value of $N_{max} = 20$, resulting in a $q_{max} = |\boldsymbol{q}|_{max} = \sqrt{3}\frac{2\pi N_{max}}{L}$, 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.

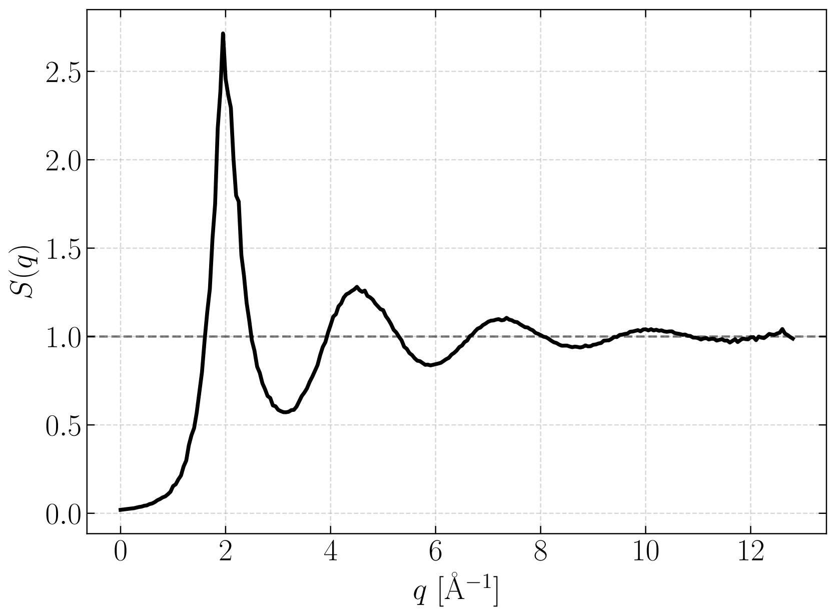

Figure 11: The static structure factor $S(q)$ for liquid aluminum at $T = 972\,\mathrm{K}$.

Conclusion

To summarize, the dynamics of an aluminum lattice with 256 atoms have been simulated using `C`. The system was in general run with $\Delta t = 0.001\,\mathrm{ps}$, 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%93Pasta%E2%80%93Ulam%E2%80%93Tsingou_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. [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.

[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. [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. [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.

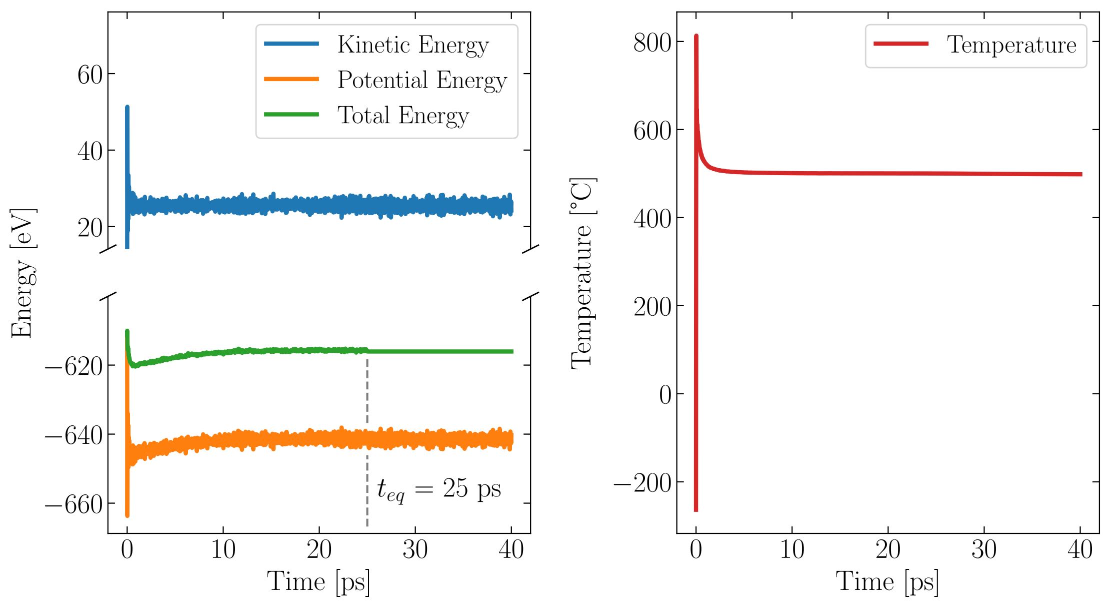

Figure A.1: The plot to the left contain instantaneous kinetic, potential and total energy for the solid system at each time step. The right plot displays the average temperature throughout the simulation. Both plots are for both equilibrium (25 ps) and production (15 ps).

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.

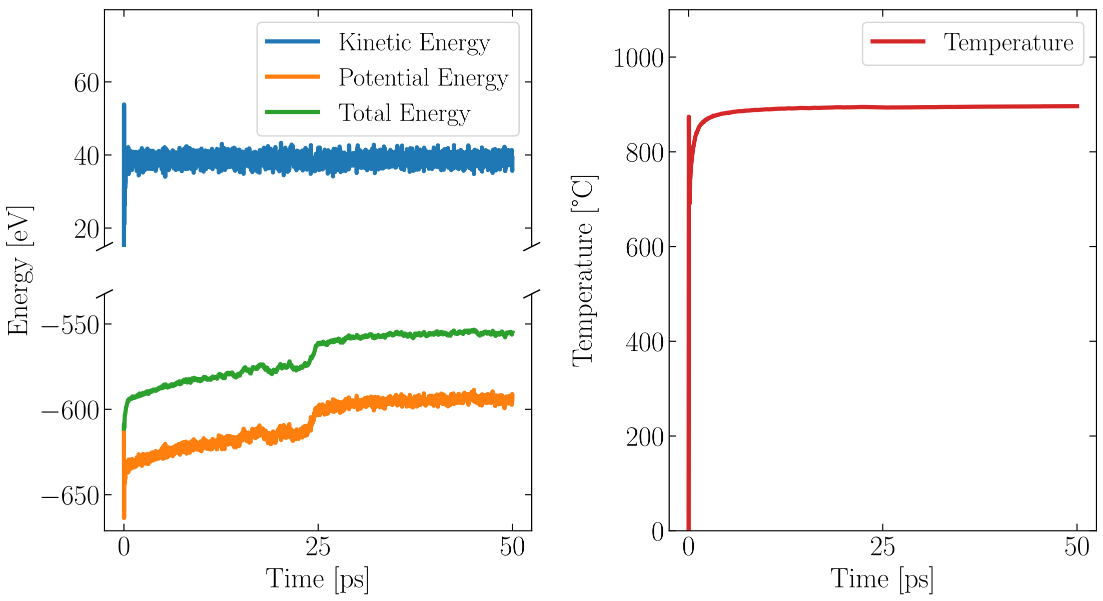

Figure A.2: The plot to the left contain instantaneous kinetic, potential and total energy for the liquid system during the warm-up phase at each time step. The right plot displays the average temperature throughout the simulation.

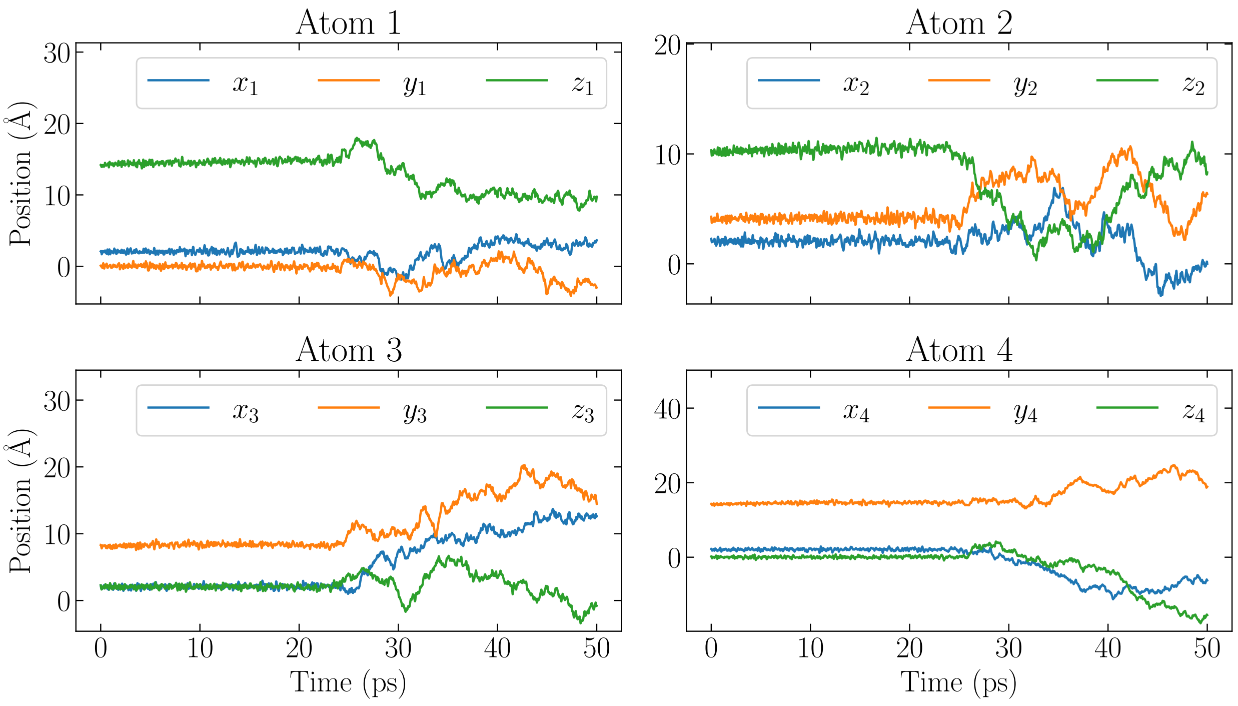

Figure A.3: Atomic trajectories: x-, y- and z coordinates, for four atoms as a function of time.

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.

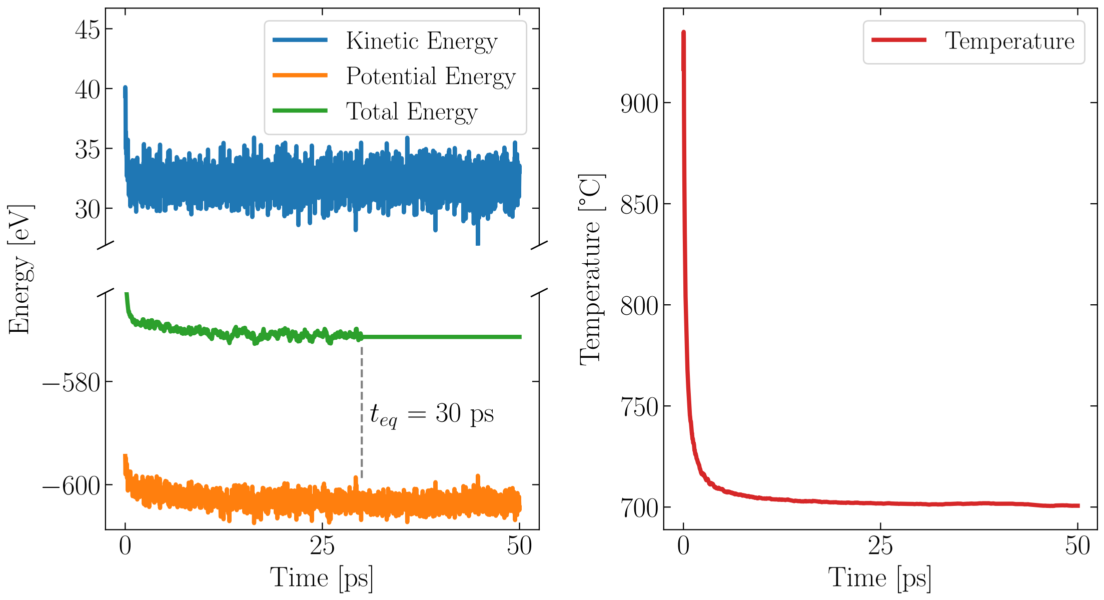

Figure A.4: The plot to the left contain instantaneous kinetic, potential and total energy for the liquid system during the production phase at each time step. The right plot displays the average temperature throughout the simulation.

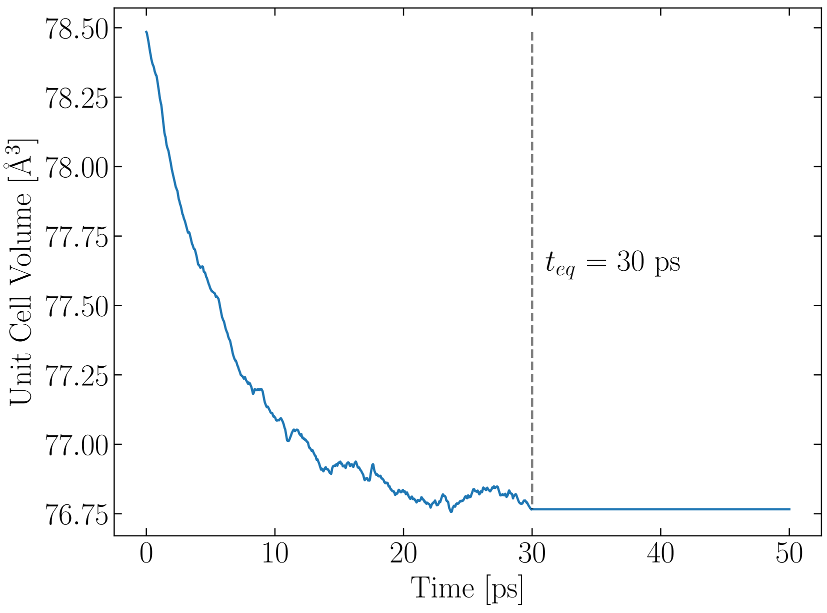

Figure A.5: Unit cell volume during the equilibration ($t < t_{eq} = 30\,\mathrm{ps}$) and production ($t > t_{eq}$) simulation runs.

B Source code in C

B.1 run.c

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

int
run(int argc, char *argv[])
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <delta_t>\n", argv[0]);
        return 1;
    }
    // _____________________________________________________________________________ //
    // ******************************** Constants ********************************** //
    // _____________________________________________________________________________ //
    int N = 256; // Number of atoms
    int cell_length = 4; // Number of unit cells in each direction
    double k_B = 8.617333262145 * 1e-5; // Boltzmann constant [eV/K]
    double m = 27.0 / 9649; // Aluminium mass [eV ps²/Å²]
    double beta = 1. / (76e3); // Bulk Modulus inverse [1 / MPa]

    // _____________________________________________________________________________ //
    // ********************************** Task 1 *********************************** //
    // _____________________________________________________________________________ //
    // Lattice parameters
    double a0[] = {4.0, 4.005, 4.01, 4.015, 4.02, 4.025, 4.03, 4.035, 4.04, 4.045,
                   4.05, 4.055, 4.06, 4.065, 4.07, 4.075, 4.08};

    // Energy file
    char *filename = "data/task_1/energies.csv";
    FILE *fp = fopen(filename, "w");

    // Iterate over the lattice parameters and calculate the energy
    for (int i = 0; i < 17; i++)
    {
        // Positions vector
        double **pos = create_2D_array(N, 3);
        // Initialize the fcc lattice
        init_fcc(pos, 4, a0[i]);
        // Calculate the energy
        double energy = get_energy_AL(pos, a0[i] * cell_length, N);
        // Write the energy to the file
        fprintf(fp, "%f, %f\n", a0[i], energy);
        // Free the memory
        destroy_2D_array(pos);
    }

    fclose(fp);

    // ____________________________________________________________________________ //
    // ********************* Initialize Task 2 & 3 & 4 ************************** //
    // ____________________________________________________________________________ //
    int t_max = 50; // The simulation time [ps]
    const double delta_t = atof(argv[1]); // The time step [ps]
    int its = (int)(t_max / delta_t); // Number of iterations

    double **positions = create_2D_array(N, 3); // Atom positions
    double **forces = create_2D_array(N, 3); // Forces
    double **velocities = create_2D_array(N, 3); // Velocities
    // Initialize the velocities to 0
    for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                velocities[i][j] = 0.0;
            }
        }

    // Initialize the random number generator
    gsl_rng *r = init_gsl_rng(42);

    double a_0 = 4.03; // Lattice constant at 0 K [Å]
    init_fcc(positions, 4, a_0); // Initialize lattice
    for (int i = 0; i < N; i++)
        {
            double rng = 0.935 + 0.13 * gsl_rng_uniform(r); // Disturbance
            // Add the disturbance to all initial atom positions
            addition_with_constant(positions[i], positions[i], a_0 * (1 - rng), 3);
        }

    // ____________________________________________________________________________ //
    // ********************************* Task 2 ********************************** //
    // ____________________________________________________________________________ //
    int its_eq = 0; // Number of iterations for equilibration, 0 for task 2

    char filename[50];
    // Format the filename with delta_t, its and its_eq
    sprintf(filename, "data/task_2/data_%.3f_%i_%i.csv", delta_t, its, its_eq);
    FILE *fp = fopen(filename, "w");
    // Write the header
    fprintf(fp, "its, t_max, delta_t, its_eq, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i,
    ↪  %i\n", its, t_max, delta_t, its_eq, 0, 0, 0, 0);
    fprintf(fp, "E_kin [eV], E_pot [eV], E_tot [eV], <T> [K], T [K], <P> [MPa], P [MPa],
    ↪  a [Å]\n");

    // Perform the Verlet algorithm
    a_0 = verlet(positions, velocities, forces, its, its_eq, delta_t, m, k_B, beta, a_0,
    ↪  N, cell_length, 0., 0., fp, NULL, NULL, NULL);

    // ____________________________________________________________________________ //
    // ********************************* Task 3 ********************************** //
    // ____________________________________________________________________________ //
    double T_eq = 500. + 273.15; // Temperature at equilibrium [K]
    double P_eq = 0.1; // Pressure at equilibrium [MPa]
    int its_eq = 25000; // Number of iterations for equilibration

    char filename[50];
    // Format the data filename with delta_t, its and its_eq
    sprintf(filename, "data/task_3/data_%.3f_%i_%i.csv", delta_t, its, its_eq);
    FILE *fp = fopen(filename, "w");
    // Write the data header
    fprintf(fp, "its, t_max, delta_t, its_eq, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i\n", its, t_max, delta_t, its_eq, 0, 0, 0, 0);
    fprintf(fp, "E_kin [eV], E_pot [eV], E_tot [eV], <T> [K], T [K], <P> [MPa], P [MPa], a [Å]\n");

    // Format the trajectory filename with delta_t, its and its_eq
    sprintf(filename, "data/task_3/trajs_%.3f_%i_%i.csv", delta_t, its, its_eq);
    FILE *fp_traj = fopen(filename, "w");
    // Write the trajectory header
    fprintf(fp_traj, "its, t_max, delta_t, its_eq, -, -, -, -, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i, %i, %i, %i, %i\n", its, t_max, delta_t, its_eq, 0, 0, 0, 0, 0, 0, 0, 0);
    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");

    // Perform the Verlet algorithm
    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, fp_traj, NULL, NULL);

    // _____________________________________________________________________________ //
    // ********************************** Task 4 *********************************** //
    // _____________________________________________________________________________ //
    int its_eq = 50000; // Number of iterations for first equilibration phase, task 4
    double T_eq = 900. + 273.15; // Temperature at first equilibrium [K]
    double P_eq = 0.1; // Pressure at first equilibrium [MPa]

    char filename[50];
    // Format the data filename with delta_t, its and its_eq
    sprintf(filename, "data/task_4/data_%.3f_%i_%i.csv", delta_t, its, its_eq);
    FILE *fp_1 = fopen(filename, "w");
    // Write the data header
    fprintf(fp_1, "its, t_max, delta_t, its_eq, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i\n", its, t_max, delta_t, its_eq, 0, 0, 0, 0);
    fprintf(fp_1, "E_kin [eV], E_pot [eV], E_tot [eV], <T> [K], T [K], <P> [MPa], P [MPa], a [Å]\n");

    // Format the trajectory filename with delta_t, its and its_eq
    sprintf(filename, "data/task_4/trajs_%.3f_%i_%i.csv", delta_t, its, its_eq);
    FILE *fp_2 = fopen(filename, "w");
    // Write the trajectory header
    fprintf(fp_2, "its, t_max, delta_t, its_eq, -, -, -, -, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i, %i, %i, %i, %i\n", its, t_max, delta_t, its_eq, 0, 0, 0, 0, 0, 0, 0, 0);
    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");

    // Perform the Verlet algorithm for the first equilibration phase
    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_1, fp_2, NULL, NULL);

    t_max = 50; // The simulation time for the second phase [ps]
    its = (int)(t_max / delta_t); // Number of iterations for the second phase
    its_eq = 30000; // Number of iterations for the second equilibration phase
    T_eq = 700. + 273.15; // Temperature at the second equilibrium [K]

    // Format the data filename with delta_t, its and its_eq
    sprintf(filename, "data/task_4/data_%.3f_%i_%i.csv", delta_t, its, its_eq);
    FILE *fp_3 = fopen(filename, "w");
    // Write the data header
    fprintf(fp_3, "its, t_max, delta_t, its_eq, -, -, -, -\n%i, %i, %f, %i, %i, %i, %i, %i\n", its, t_max, delta_t, its_eq, 0, 0, 0, 0);
    fprintf(fp_3, "E_kin [eV], E_pot [eV], E_tot [eV], <T> [K], T [K], <P> [MPa], P [MPa], a [Å]\n");

    // Format the trajectory filename with delta_t, its and its_eq
    sprintf(filename, "data/task_4/trajs_%.3f_%i_%i.csv", delta_t, its, its_eq);
    FILE *fp_4 = fopen(filename, "w");
    // Write the trajectory header
    fprintf(fp_4, "its, t_max [ps], delta_t [ps], its_eq, T_eq [K], P_eq [MPa], -, -, -, -, -, -\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);
    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");

    // Format the radial distribution filename with delta_t, its and its_eq
    sprintf(filename, "data/task_4/rdist_%.3f_%i_%i.csv", delta_t, its, its_eq);
    FILE *fp_rdist = fopen(filename, "w");
    // Write the radial distribution header
    fprintf(fp_rdist, "its, its_eq, delta_t [ps], a_0 [Å]\n");

    // Format the structure factor filename with delta_t, its and its_eq
    sprintf(filename, "data/task_4/sfact_%.3f_%i_%i.csv", delta_t, its, its_eq);
    FILE *fp_sfact = fopen(filename, "a");
    // Write the structure factor header
    fprintf(fp_sfact, "its, its_eq, delta_t [ps], a_0 [Å]\n");

    // Perform the Verlet algorithm for the second phase
    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);

    // Close files and free memory
    fclose(fp);
    fclose(fp_traj);
    fclose(fp_1);
    fclose(fp_2);
    fclose(fp_3);
    fclose(fp_4);
    fclose(fp_rdist);
    fclose(fp_sfact);
    gsl_rng_free(r);
    destroy_2D_array(positions);
    destroy_2D_array(forces);
    destroy_2D_array(velocities);

    return 0;
}

gsl_rng *
init_gsl_rng(int seed){
    const gsl_rng_type * T;
    gsl_rng * r;
    gsl_rng_env_setup();
    T = gsl_rng_default; // default random number generator
    r = gsl_rng_alloc(T); // allocate memory for the random number generator

    if (!r) {
        fprintf(stderr, "Error: Could not allocate memory for RNG.\n");
        exit(EXIT_FAILURE); // Exit if allocation fails
    }

    // Set the seed
    gsl_rng_set(r, seed);

    return r;
}

double
verlet(double **positions, double **velocities, double **forces, int its, int its_eq,
            double delta_t, double m, double k_B, double beta, double a_0, int N,
            int cell_length, double T_eq, double P_eq, FILE *fp, FILE *fp_traj,
            FILE *fp_rdist, FILE *fp_sfact)
{
    double tau_T = delta_t * 200; // Time constant for temperature [ps]
    double tau_P = delta_t * 1000; // Time constant for pressure [ps]
    double alpha_T; // Scaling factor for temperature
    double alpha_P; // Scaling factor for pressure
    double E_pot = 0.0; // Potential energy (instantaneous)
    double E_kin = 0.0; // Kinetic energy (instantaneous)
    double virial = 0.0; // Virial
    double *P = malloc(its * sizeof(double)); // Instantaneous pressure
    double *T = malloc(its * sizeof(double)); // Instantaneous temperature
    double T_avg = 0.0; // Average temperature
    double P_avg = 0.0; // Average pressure

    // Calculate the initial forces, potential energy and virial
    calculate(&E_pot, &virial, forces, positions, a_0 * cell_length, N);
    for (unsigned int i = 0; i < its; i++)
    {
        E_kin = 0.0;
        // Perform the first half step
        for (unsigned int j = 0; j < N; j++)
        {
            for (unsigned int k = 0; k < 3; k++)
            {
                velocities[j][k] += 0.5 * delta_t * forces[j][k] / m;
                positions[j][k] += delta_t * velocities[j][k];
            }
        }
        // Calculate new accelerations
        calculate(&E_pot, &virial, forces, positions, a_0 * cell_length, N);
        // Perform the second half step
        for (unsigned int j = 0; j < N; j++)
    {
            for (unsigned int k = 0; k < 3; k++)
            {
                velocities[j][k] += 0.5 * delta_t * forces[j][k] / m;
            }
            E_kin += 0.5 * m * vector_norm(velocities[j], 3) * vector_norm(velocities[j],
             ↪  3);
        }

        T[i] = 2.0 / 3.0 / k_B / N * E_kin; // Calculate the instantaneous temperature
        ↪   [K]
        T_avg = average(T, i+1); // Calculate the average temperature [K]
        P[i] = 1 / (3 * 64 * pow(a_0, 3)) * (E_kin + virial) / 6.2415 * 1e6; // Calculate
        ↪   the instantaneous pressure [MPa]
        P_avg = average(P, i+1); // Calculate the average pressure [MPa]

        // Scale the temperature and pressure if the equilibration phase is not over
        if (i < its_eq)
        {
            // Calculate the scaling factors
            alpha_T = 1 + 2 * delta_t * (T_eq - T[i]) / (T[i] * tau_T);
            alpha_P = 1 - beta * delta_t * (P_eq - P[i]) / tau_P;

            // Scale the lattice constant
            a_0 = a_0 * pow(alpha_P, 1. / 3.);

            // Scale the temperature and velocities
            for (unsigned int j = 0; j < N; j++)
            {
                multiplication_with_constant(positions[j], positions[j], pow(alpha_P, 1.
                ↪   / 3.), 3);
                multiplication_with_constant(velocities[j], velocities[j], sqrt(alpha_T),
                ↪   3);
            }
        }
        // Write the data (energy, temperature, pressure and lattice constant) to the
        ↪   data file
        if (fp != NULL)
        {
            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);
        }
        // Write the trajectory data to the trajectory file
        if (fp_traj != NULL)
        {
            fprintf(fp_traj, "%f, %f, %f, %f, %f, %f, %f, %f, %f, %f, %f, %f\n",
                            positions[15][0], positions[15][1], positions[15][2],
                            positions[27][0], positions[27][1], positions[27][2],
                            positions[35][0], positions[35][1], positions[35][2],
                            positions[49][0], positions[49][1], positions[49][2]);
        }
        // Calculate the radial distribution and write it to the radial distribution file
        if (fp_rdist != NULL && i >= its_eq)
        {
            double bin_width = 0.05; // Width of the bins
            double L = a_0 * 4; // Length of the supercell
            int N_bins = (int)(L / 2 / bin_width); // Number of bins
            double *bins = calloc(N_bins, sizeof(double)); // Bins for the radial
            ↪   distribution
            // Write the header
            if (i == its_eq){
                fprintf(fp_rdist, "%i, %i, %f, %f\n", its, its_eq, delta_t, a_0);
            }

            // Calculate the radial distribution
            radial_dist(bins, positions, N_bins, bin_width, N, L);

            // Write the radial distribution to the file
            for (int j = 0; j < N_bins; j++)
            {
                if (j == N_bins - 1)
                {
                    fprintf(fp_rdist, "%f", bins[j]);
                }
                else
                {
                    fprintf(fp_rdist, "%f, ", bins[j]);
                }
            }
            fprintf(fp_rdist, "\n");
        }
        // Calculate the structure factor and write it to the structure factor file
        if (fp_sfact != NULL && i >= its_eq)
        {
            int N_max = 10; // Maximum value in each direction for n_x, n_y and n_z
            double L = a_0 * 4; // Length of the supercell
            int n = 2*N_max + 1; // Number of grid points in each direction
            int n_points = n * n * n; // Total number of grid points
            double *S_q = (double *)malloc(n_points * sizeof(double)); // Structure
            ↪  factor
            int n_bins = 500; // Number of bins
            double q_max = 2 * M_PI / L * N_max * sqrt(3); // Maximum value of the
            ↪  magnitude of q
            double bin_width = q_max / n_bins; // Width of the bins

            // Initialize the grid and keep it throughout the iterations
            static double ****grid = NULL;
            if (i == its_eq)
            {
                grid = init_grid(N_max, L);
            }
            // Calculate the structure factor
            structure_factor(grid, positions, S_q, N, n);
            // Calculate the spherical average of the structure factor
            spherical_avg(grid, S_q, n, n_bins, bin_width, fp_sfact);
            // Free memory
            if (i == its - 1)
            {
                destroy_grid(grid, N_max);
                free(S_q);
            }
        }
    }
    // Free memory
    free(T);
    free(P);

    return a_0;
}
void
radial_dist(double *bins, double **positions, int N_bins, double bin_width, int N, double
↪  L)
```

```c
{
    double r; // Distance between atoms
    double V = L * L * L; // Volume of the supercell
    double norm_factor; // Normalization factor
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if (i != j)
            {
                // Calculate the distance between the atoms
                r = boundary_distance_between_vectors(positions[i], positions[j], 3, L);
                for (int l = 0; l < N_bins; l++)
                {
                    // Check if the distance is within the bin, and increment the
                    // bin if true
                    if (l * bin_width < r && r < (l + 1) * bin_width)
                    {
                        bins[l] += 1;
                    }
                }
            }
        }
    }
    for (int l = 0; l < N_bins; l++)
    {
        bins[l] /= (N - 1); // Average over the number of atoms
        norm_factor = (N - 1) * 4 * M_PI * (3 * pow((l + 1), 2) - 3 * (l + 1) + 1) *
        ↪  pow(bin_width, 3) / 3 / V;
        bins[l] /= norm_factor; // Apply the scale factor
    }
}
double
boundary_distance_between_vectors(double *v1, double *v2, int dim, double box_length)
{
    double r = 0.0;
    double delta;
    for (int d = 0; d < dim; d++)
    {
        delta = v1[d] - v2[d];
        // Apply boundary conditions
        delta -= round(delta / box_length) * box_length;
        r += delta * delta;
    }

    return sqrt(r);
}
double ****
init_grid(int N_max, double L)
{
    int N = 2*N_max + 1; // Number of grid points in each direction
    double ****grid = (double ****)malloc(N * sizeof(double ***)); // Initialize the grid
    for (int i = 0; i < N; i++)
    {
        // Allocate memory for the grid
        grid[i] = (double ***)malloc(N * sizeof(double **));
        for (int j = 0; j < N; j++)
        {
            grid[i][j] = (double **)malloc(N * sizeof(double *));
            for (int k = 0; k < N; k++)
            {
                grid[i][j][k] = (double *)malloc(3 * sizeof(double));
                // Calculate the grid points
                grid[i][j][k][0] = 2 * M_PI / L * (i - N_max);
                grid[i][j][k][1] = 2 * M_PI / L * (j - N_max);
                grid[i][j][k][2] = 2 * M_PI / L * (k - N_max);
            }
        }
    }
    return grid;
}
void
destroy_grid(double ****grid, int N_max)
{
    int N = 2*N_max + 1;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            for (int k = 0; k < N; k++)
            {
                free(grid[i][j][k]);
            }
            free(grid[i][j]);
        }
        free(grid[i]);
    }
    free(grid);
}
void
structure_factor(double ****grid, double **positions, double *Sq, int N, int n)
{
    double S_q_cos; // S(q) cosine term
    double S_q_sin; // S(q) sine term
    int it = 0; // Iterator
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            for (int k = 0; k < n; k++)
            {
                // The q vector components
                double qx = grid[i][j][k][0];
                double qy = grid[i][j][k][1];
                double qz = grid[i][j][k][2];

                S_q_cos = 0.0;
                S_q_sin = 0.0;
                // Calculate dot product for each atom
                for (int l = 0; l < N; l++)
                {
                    double dot_product = (positions[l][0] * qx +
                                          positions[l][1] * qy +
                                          positions[l][2] * qz);
                    S_q_cos += cos(dot_product);
                    S_q_sin += sin(dot_product);
                }
                // Calculate the structure factor at the given grid point
                Sq[it] = 1.0 / N * (S_q_cos * S_q_cos + S_q_sin * S_q_sin);
                it++; // Increment the iterator
            }
        }
    }
}
void
spherical_avg(double ****grid, double *Sq, int n, int n_bins, double bin_width, FILE *fp_sfact)
{
    double *S_avg = calloc(n_bins, sizeof(double)); // Spherical average
    int *counts = calloc(n_bins, sizeof(double)); // Counts for each bin
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            for (int k = 0; k < n; k++)
            {
                // Calculate the magnitude of the q vector
                double q = sqrt(grid[i][j][k][0] * grid[i][j][k][0] +
                                grid[i][j][k][1] * grid[i][j][k][1] +
                                grid[i][j][k][2] * grid[i][j][k][2]);
                int bin = (int)(q / bin_width); // Bin index
                // Bin index cannot exceed the number of bins
                if (bin == n_bins){
                    bin = n_bins - 1;
                }
                // Calculate the spherical average and increment the count
                S_avg[bin] += Sq[i * n * n + j * n + k];
                counts[bin]++;
            }
        }
    }
    // Write the spherical average to the file if the bin is not empty
    for (int b = 0; b < n_bins; b++) {
        if (counts[b] > 0)
        {
            if (b == n_bins - 1){
                fprintf(fp_sfact, "%f\n", S_avg[b] / counts[b]);
            }
            else{
                fprintf(fp_sfact, "%f, ", S_avg[b] / counts[b]);
            }
        }
    }
    free(S_avg);
    free(counts);
}
```

B.2 tools.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gsl/gsl_fft_real.h>
#include <gsl/gsl_fft_halfcomplex.h>
#include <complex.h>

#include "tools.h"

void
elementwise_addition(
                     double *res,
                     double *v1,
                     double *v2,
                     unsigned int len
                    )
{
    for (int i = 0; i < len; i++)
    {
        res[i] = v1[i] + v2[i];
    }
}

void
elementwise_multiplication(
                           double *res,
                           double *v1,
                           double *v2,
                           unsigned int len
                          )
{
    for (int i = 0; i < len; i++)
    {
        res[i] = v1[i] * v2[i];
    }
}

void
addition_with_constant(
                       double *res,
                       double *v,
                       double constant,
                       unsigned int len)
{
    for (int i = 0; i < len; i++)
    {
        res[i] = v[i] + constant;
    }
}

void
multiplication_with_constant(
                             double *res,
                             double *v,
                             double constant,
                              unsigned int len)
{
    for (int i = 0; i < len; i++)
    {
        res[i] = v[i] * constant;
    }
}

double
dot_product(
            double *v1,
            double *v2,
            unsigned int len
           )
{
    double res = 0.;
    for (int i = 0; i < len; i++)
    {
        res += v1[i] * v2[i];
    }
    return res;
}

double **
create_2D_array(
                unsigned int row_size,
                unsigned int column_size
               )
{
    // Allocate memory for row pointers
    double **array = malloc(row_size * sizeof(double *));
    if (array == NULL)
    {
        fprintf(stderr, "Memory allocation failed for row pointers.\n");
        return NULL;
    }

    // Allocate a single contiguous block for all elements, i.e. all matrix elements
    // get stored in a single block of memory, array[0] is the start of the block
    // i.e. here we have array[0][index] as the way to access the elements
    array[0] = malloc(row_size * column_size * sizeof(double));
    if (array[0] == NULL)
    {
        fprintf(stderr, "Memory allocation failed for data block.\n");
        free(array);
        return NULL;
    }

    // Set the row pointers to the appropriate positions in the block, in order to
    // allow for the array[i][j] syntax. Now array[i] points to the start of the
    // i-th row.
    for (int i = 1; i < row_size; i++)
    {
        array[i] = array[0] + i * column_size;
    }

    return array;
```

```c
}

void
destroy_2D_array(
                 double **array
                )
{
    if (array != NULL)
    {
        free(array[0]); // Free the contiguous block
        free(array); // Free the row pointers
    }
}

void
matrix_vector_multiplication(
                             double *result,
                             double **A,
                             double *b,
                             unsigned int n,
                             unsigned int m
                            )
{
    for (int i = 0; i < n; i++)
    {
        result[i] = 0.;
        for (int j = 0; j < m; j++)
        {
            result[i] += A[i][j] * b[j];
        }
    }
}

void
matrix_matrix_multiplication(
                             double **result,
                             double **A,
                             double **B,
                             unsigned int n,
                             unsigned int m,
                             unsigned int k
                            )
{
    for (int i = 0; i < n; i++)
    {
        for (int kappa = 0; kappa < k; kappa++)
        {
            result[i][kappa] = 0.;
            for (int j = 0; j < m; j++)
            {
                result[i][kappa] += A[i][j] * B[j][kappa];
            }
        }
    }
}

double
```

```c
vector_norm(
            double *v,
            unsigned int len
           )
{
    double res = 0.;
    for (int i = 0; i < len; i++)
    {
        res += v[i] * v[i];
    }
    return sqrt(res);
}


void
normalize_vector(
                 double *v,
                 unsigned int len
                )
{
    double norm = vector_norm(v, len);
    multiplication_with_constant(v, v, 1. / norm, len);
}

double
average(
        double *v,
        unsigned int len
       )
{
    double res = 0.;
    for (int i = 0; i < len; i++)
    {
        res += v[i];
    }
    return res / len;
}


double
standard_deviation(
                       double *v,
                       unsigned int len
                  )
{
    double avg = average(v, len);
    double res = 0.;
    for (int i = 0; i < len; i++)
    {
        res += (v[i] - avg) * (v[i] - avg);
    }
    return sqrt(res / len);
}

double
distance_between_vectors(
                         double *v1,
                            double *v2,
                            unsigned int len
                           )
{
    double res = 0.;
    // With previous defined functions
    double *diff = malloc(len * sizeof(double));
    multiplication_with_constant(v1, v1, -1., len);
    elementwise_addition(diff, v1, v2, len);
    res = vector_norm(diff, len);
    free(diff);
    return res;
}

void
cumulative_integration(
                        double *res,
                        double *v,
                        double dx,
                        unsigned int v_len
                      )
{
    double sum = 0.;
    res[0] = 0.;
    for (int i = 1; i < v_len; i++)
    {
        sum = 0.5 * (v[i - 1] + v[i]) * dx;
        res[i] = res[i - 1] + sum;
    }
}

void
write_xyz(
          FILE *fp,
          char *symbol,
          double **positions,
          double **velocities,
          double alat,
          int natoms)
{
    fprintf(fp, "%i\nLattice=\"%f 0.0 0.0 0.0 %f 0.0 0.0 0.0 %f\" ", natoms, alat, alat,
    ↪  alat);
    fprintf(fp, "Properties=species:S:1:pos:R:3:vel:R:3 pbc=\"T T T\"\n");
    for(int i = 0; i < natoms; ++i)
    {
        fprintf(fp, "%s %f %f %f %f %f %f\n",
                symbol, positions[i][0], positions[i][1], positions[i][2],
                velocities[i][0], velocities[i][1], velocities[i][2]);
    }
}

void fft_freq(
          double *res,
              int n,
              double timestep)
{
    for (int i = 0; i < n; i++)
    {
        if (i < n / 2)
        {
            res[i] = 2 * M_PI * i / (n * timestep);
        }
        else
        {
            res[i] = 2 * M_PI * (i - n) / (n * timestep);
        }
    }
}

/* Freely given functions */
void
skip_line(FILE *fp)
{
    int c;
    while (c = fgetc(fp), c != '\n' && c != EOF);
}

void
read_xyz(
         FILE *fp,
         char *symbol,
         double **positions,
         double **velocities,
         double *alat)
{
    int natoms;
    if(fscanf(fp, "%i\nLattice=\"%lf 0.0 0.0 0.0 %lf 0.0 0.0 0.0 %lf\" ", &natoms, alat,
    ↪  alat, alat) == 0){
        perror("Error");
    }
    skip_line(fp);
    for(int i = 0; i < natoms; ++i)
    {
        fscanf(fp, "%s %lf %lf %lf ",
                symbol, &positions[i][0], &positions[i][1], &positions[i][2]);
        fscanf(fp, "%lf %lf %lf\n",
                &velocities[i][0], &velocities[i][1], &velocities[i][2]);
    }
}

void powerspectrum(
           double *res,
           double *signal,
           int n,
                   double timestep)
{
    /* Declaration of variables */
    double *complex_coefficient = malloc(sizeof(double) * 2*n); // array for the complex
    ↪  fft data
    double *data_cp = malloc(sizeof(double) * n);

    /*make copy of data to avoid messing with data in the transform*/
    for (int i = 0; i < n; i++)
    {
        data_cp[i] = signal[i];
    }

    /* Declare wavetable and workspace for fft */
    gsl_fft_real_wavetable *real;
    gsl_fft_real_workspace *work;

    /* Allocate space for wavetable and workspace for fft */
    work = gsl_fft_real_workspace_alloc(n);
    real = gsl_fft_real_wavetable_alloc(n);

    /* Do the fft*/
    gsl_fft_real_transform(data_cp, 1, n, real, work);

    /* Unpack the output into array with alternating real and imaginary part */
    gsl_fft_halfcomplex_unpack(data_cp, complex_coefficient,1,n);

    /*fill the output powspec_data with the powerspectrum */
    for (int i = 0; i < n; i++)
    {
        res[i] = (complex_coefficient[2*i]*complex_coefficient[2*i]+
                  complex_coefficient[2*i+1]*complex_coefficient[2*i+1]);
        res[i] *= timestep / n;
    }

    /* Free memory of wavetable and workspace */
    gsl_fft_real_wavetable_free(real);
    gsl_fft_real_workspace_free(work);
    free(complex_coefficient);
    free(data_cp);
}
```

C Source code in Python

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.constants as K
import pandas as pd

# Latex style
plt.style.use('default')
plt.rc('text', usetex=True)
plt.rc('font', family='serif')
plt.rc('font', size=20)
plt.rcParams['text.latex.preamble'] = r'\usepackage{amsmath}'

# Set ticks on both sides
plt.rcParams['xtick.direction'] = 'in'
plt.rcParams['ytick.direction'] = 'in'
plt.rcParams['xtick.major.size'] = 5
plt.rcParams['ytick.major.size'] = 5
plt.rcParams['xtick.top'] = True
plt.rcParams['ytick.right'] = True

```

```python
# Constants
k_B = K.Boltzmann
e = K.elementary_charge
k_B /= e
n_atoms = 256

# Functions
def read_data(delta_t, its, its_eq, task, opt=None):
    if task == 2:
        filename = f'data/task_2/data_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
    elif task == 3:
        filename = f'data/task_3/data_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
    elif task == 4:
        filename = f'data/task_4/data_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
    elif task == 6:
        filename = f'data/task_4/rdist_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
    elif task == 7:
        if opt is not None:
            filename =
            ↪  f'data/task_4/sfact_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}_{opt}.csv'
        else:
            filename = f'data/task_4/sfact_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
    if task not in [6, 7]:
        data_facts = np.genfromtxt(filename, dtype=np.float64, delimiter=',', max_rows=2)
        data = np.genfromtxt(filename, dtype=np.float64, encoding=None,
        ↪  delimiter=',')[3:, :]
        its, t_max, delta_t, its_eq, _, _, _, _ = data_facts[1]
        a_0, N_bins, N_max = None, None, None
    elif task == 6:
        data_facts = np.genfromtxt(filename, dtype=np.float64, delimiter=',', max_rows=2)
        data = np.genfromtxt(filename, dtype=np.float64, encoding=None, delimiter=',',
        ↪  skip_header=2)
        N_bins = data.shape[1]
        its, its_eq, delta_t, a_0 = data_facts[1]
        t_max, N_max = None, None
    elif task == 7:
        data_facts = np.genfromtxt(filename, dtype=np.float64, delimiter=',', max_rows=2)
        data = np.genfromtxt(filename, dtype=np.float64, encoding=None, delimiter=',',
        ↪  skip_header=2)
        N_bins = data.shape[1] - 1
        its, its_eq, delta_t, a_0, N_max, _ = data_facts[1]
        t_max = None

    return data, int(its), t_max, delta_t, int(its_eq), N_bins, a_0, N_max

def plot_trajs(its_eq, its, t_max, delta_t, task, save=True):

    def read_trajs():
        if task == 3:
            filename = f'data/task_3/trajs_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
        elif task == 4:
            filename = f'data/task_4/trajs_{delta_t:.3f}_{its:.0f}_{its_eq:.0f}.csv'
        trajs = np.genfromtxt(filename, dtype=np.float64, encoding=None, delimiter=',')

        return trajs[3:, :]


    trajs = read_trajs()

    fig, axs = plt.subplots(2, 2, figsize=(12, 7), sharex=True)

    its_eq = 0
    t = np.linspace(0, t_max - t_max*its_eq/its, int(its-its_eq))

    axs[0][0].plot(t, trajs[its_eq:, 0], label='$x_1$')
    axs[0][0].plot(t, trajs[its_eq:, 1], label='$y_1$')
    axs[0][0].plot(t, trajs[its_eq:, 2], label='$z_1$')
    axs[0][0].set_title('Atom 1')

    axs[0][1].plot(t, trajs[its_eq:, 3], label='$x_2$')
    axs[0][1].plot(t, trajs[its_eq:, 4], label='$y_2$')
    axs[0][1].plot(t, trajs[its_eq:, 5], label='$z_2$')
    axs[0][1].set_title('Atom 2')

    axs[1][0].plot(t, trajs[its_eq:, 6], label='$x_3$')
    axs[1][0].plot(t, trajs[its_eq:, 7], label='$y_3$')
    axs[1][0].plot(t, trajs[its_eq:, 8], label='$z_3$')
    axs[1][0].set_title('Atom 3')

    axs[1][1].plot(t, trajs[its_eq:, 9], label='$x_4$')
    axs[1][1].plot(t, trajs[its_eq:, 10], label='$y_4$')
    axs[1][1].plot(t, trajs[its_eq:, 11], label='$z_4$')
    axs[1][1].set_title('Atom 4')

    for ax in axs.flatten():
        ax.legend(loc='upper right', ncol=3)
        y_min, y_max = ax.get_ylim()
        ax.set_ylim(y_min, y_max + 0.5 * (y_max - y_min))

    axs[0][0].set_ylabel('Position (Å)')
    axs[1][0].set_ylabel('Position (Å)')
    axs[1][0].set_xlabel('Time (ps)')
    axs[1][1].set_xlabel('Time (ps)')

    plt.tight_layout()

    return fig

def plot_T_E(data, its, its_eq, t_max, delta_t, save=True):
    def init_TE_fig():
        fig, axs = plt.subplots(2, 2, figsize=(12, 7), sharex=True)
        fig.subplots_adjust(hspace=0.05)  # adjust space between Axes
        ax1, ax2, ax3, ax4 = axs[0][0], axs[1][0], axs[0][1], axs[1][1]
        ax3.set_visible(False)
        ax4.set_visible(False)

        ax1_2 = fig.add_subplot(1, 2, 1, frameon=False)
        ax1_2.set_ylabel('Energy [eV]', labelpad=50)
        ax1_2.set_xticks([])
        ax1_2.set_yticks([])

        ax3 = fig.add_subplot(1, 2, 2)
        ax3.set_ylabel('Temperature [°C]', labelpad=10)
        plt.suptitle(f'\\textbf{{Energies and Temperature vs. time $\\vert
        ↪  \\hspace{{0.3cm}}\\Delta t = {delta_t}$ ps}}')

        d = .5  # proportion of vertical to horizontal extent of the slanted line
        kwargs = dict(marker=[(-1, -d), (1, d)], markersize=12,
                      linestyle="none", color='k', mec='k', mew=1, clip_on=False)
        ax1.plot([0, 1], [0, 0], transform=ax1.transAxes, **kwargs)
        ax2.plot([0, 1], [1, 1], transform=ax2.transAxes, **kwargs)

        ax1.spines.bottom.set_visible(False)
        ax2.spines.top.set_visible(False)

        return fig, ax1, ax2, ax3, ax4

    t = np.linspace(0, t_max, int(its))
    E_k_min, E_k_max = np.min(data[1:, 0]), np.max(data[1:, 0])
    E_p_min, E_p_max = np.min(data[1:, 1]), np.max(data[1:, 1])

    fig, ax1, ax2, ax3, ax4 = init_TE_fig()
    ax1.plot(t, data[:, 0], label='Kinetic Energy', linewidth=3)
    ax1.plot(t, data[:, 1], label='Potential Energy', linewidth=3)
    ax1.plot(t, data[:, 2], label='Total Energy', linewidth=3)
    ax2.plot(t, data[:, 0], label='Kinetic Energy', linewidth=3)
    ax2.plot(t, data[:, 1], label='Potential Energy', linewidth=3)
    ax2.plot(t, data[:, 2], label='Total Energy', linewidth=3)
    ax1.set_ylim(E_k_max - 0.75 * (E_k_max - E_k_min), E_k_max + 0.5 * (E_k_max -
    ↪  E_k_min))
    ax2.set_ylim(E_p_min - 0.1 * (E_p_max - E_p_min), E_p_max + 0.25 * (E_p_max -
    ↪  E_p_min))

    ax3.plot(t, data[:, 3] - 273.15, label='Temperature', color='tab:red', linewidth=3)
    for ax in (ax2, ax3):
        ax.set_xlabel('Time [ps]')

    ax1.tick_params(axis='x', which='both', bottom=False, top=True, labelbottom=False)
    ax2.tick_params(axis='x', which='both', bottom=True, top=False, labelbottom=True)
    ax1.legend(fontsize=18, loc='upper right')
    ax3.legend(fontsize=18, loc='upper right')
    plt.tight_layout()

    return fig

def plot_T_P(data, its_eq, save=False):
    def init_TP_fig():
        fig, ax1 = plt.subplots(1, 1, figsize=(8, 6))
        ax2 = ax1.twinx()
        ax1.set_ylabel('Temperature [°C]', labelpad=10, color='tab:red')
        ax2.set_ylabel('Pressure [MPa]', labelpad=10, color='tab:blue')


        for ax in (ax1, ax2):
            if ax == ax1:
                ax.tick_params(axis='y', direction='in', length=5, width=1,
                ↪  colors='tab:red', right=False, pad=10)
            else:
                ax.tick_params(axis='y', direction='in', length=5, width=1,
                ↪  colors='tab:blue', left=False, pad=10)

        return fig, ax1, ax2
    T_avg = data[:, 4] - 273.15 # Convert to Celsius
    P_avg = data[:, 6]
    t = np.linspace(0, t_max, its)

    fig, ax1, ax2 = init_TP_fig()
    ax1.plot(t, T_avg, color='tab:red', label='Temperature')
    ax2.plot(t, P_avg, color='tab:blue', label='Pressure')
    ax1.set_xlabel('Time [ps]')
    ax1.set_yticks([0, 500, 1000])
    ax2.set_ylim(-500, 1000)
    ax2.set_yticks([-500, 0, 500, 1000])
    ax1.legend(loc='upper left', labelcolor='tab:red')
    ax2.legend(loc='upper right', labelcolor='tab:blue')
    plt.tight_layout()

    return fig

def plot_volume_evo(data, its, t_max, delta_t, its_eq, save=True):
    volume_unit_cell = np.power(data[:, 7], 3)
    t = np.linspace(0, t_max, its)

    fig, ax = plt.subplots(figsize=(8, 6))
    plt.plot(t, volume_unit_cell)
    plt.xlabel('Time [ps]')
    plt.ylabel('Unit Cell Volume [Å$^3$]')
    plt.tight_layout()

    return fig

def save_fig(fig, name, task):
    fig.savefig(f'figs/task_{task}/{name}.pdf')

######################################## Task 1 ########################################
save = False
data = np.loadtxt('data/task_1/energies.csv', delimiter=',')

fig, ax = plt.subplots()

plt.plot(np.power(data[:, 0], 3), data[:, 1] / 64, 'x', ms=10, label='Data Points')
fit = np.polyfit(np.power(data[:, 0], 3), data[:, 1] / 64, 2)
plt.plot(np.power(data[:, 0], 3), np.polyval(fit, np.power(data[:, 0], 3)), 'k--',
↪  label='Quadratic Fit')

print(f'The unit cell volume generating the minimum energy is {np.power(-fit[1] / (2 *
↪  fit[0]),1):.3f} Å^3')
print(f'The lattice parameter value generating the minimum energy is {np.power(-fit[1] /
↪  (2 * fit[0]), 1/3):.3f} Å')

plt.xlabel('Unit Cell Volume (Å$^3$)')
plt.ylabel('Energy/unit cell (eV)')
plt.legend()
plt.tight_layout()
plt.show()

if save:
    save_fig(fig, 'energy_vs_volume', 1)
```

```python
######################################## Task 2 ########################################
t_max = 10

save = False
data_01, its_01, t_max_01, delta_t_01, _, _ , _, _ = read_data(0.1, t_max/0.1, 0, 2)
data_002, its_002, t_max_002, delta_t_002, _, _ , _, _ = read_data(0.02, t_max/0.02, 0,
↪  2)
data_0001, its_0001, t_max_0001, delta_t_0001, _, _ , _, _ = read_data(0.001,
↪  t_max/0.001, 0, 2)

fig_01 = plot_T_E(data_01, its_01, None, t_max_01, delta_t_01)
fig_002 = plot_T_E(data_002, its_002, None, t_max_002, delta_t_002)
fig_0001 = plot_T_E(data_0001, its_0001, None, t_max_0001, delta_t_0001)

if save:
    save_fig(fig_01, 'TE_plot_dt01', 2)
    save_fig(fig_005, 'TE_plot_dt005', 2)
    save_fig(fig_0001, 'TE_plot_dt0001', 2)

######################################## Task 3 ########################################
save = False
data, its, t_max, delta_t, its_eq, _, _, _ = read_data(0.001, 40000, 25000, task=3)

fig_TE = plot_T_E(data, its, its_eq, t_max, delta_t)
fig_TP = plot_T_P(data, its_eq)
fig_evo = plot_volume_evo(data, its, t_max, delta_t, its_eq)
fig_trajs = plot_trajs(its_eq, its, t_max, delta_t, task=3)

if save:
    save_fig(fig_TE, 'TE_plot', 3)
    save_fig(fig_TP, 'TP_plot', 3)
    save_fig(fig_evo, 'volume_evo', 3)
    save_fig(fig_trajs, 'trajs_3', 3)

######################################## Task 4 ########################################
save = False
task = 4
data, its, t_max, delta_t, its_eq, _, _, _ = read_data(0.001, 50000, 30000, task)

fig_trajs = plot_trajs(its_eq, its, t_max, delta_t, task)
fig_evo = plot_volume_evo(data, its, t_max, delta_t, its_eq)
fig_TP = plot_T_P(data, its_eq)
fig_TE = plot_T_E(data, its, its_eq, t_max, delta_t)

if save:
    save_fig(fig, 'trajs_4', task)
    save_fig(fig_evo, 'volume_evo', task)
    save_fig(fig_TP, 'TP_plot', task)
    save_fig(fig_TE, 'TE_plot', task)

data, its, t_max, delta_t, its_eq, _, _, _ = read_data(0.001, 50000, 50000, task)

fig_trajs = plot_trajs(its_eq, its, t_max, delta_t, task)
fig_evo = plot_volume_evo(data, its, t_max, delta_t, its_eq)
fig_TP = plot_T_P(data, its_eq)
fig_TE = plot_T_E(data, its, its_eq, t_max, delta_t)

```

```python
if save:
    save_fig(fig_trajs, 'warm_up/trajs_4_warmup', task)
    save_fig(fig_evo, 'warm_up/volume_evo_warmup', task)
    save_fig(fig_TP, 'warm_up/TP_plot_warmup', task)
    save_fig(fig_TE, 'warm_up/TE_plot_warmup', task)

######################################## Task 5 ########################################
def avg_fluct_square(data):
    data_avg = np.mean(data)
    d_data = np.mean((data - data_avg)**2)

    return d_data

def CV(data, its_eq, n_atoms, k_B, e):
    T_avg = np.mean(data[its_eq:, 4])
    dE_kin = avg_fluct_square(data[its_eq:, 0])
    dE_pot = avg_fluct_square(data[its_eq:, 1])

    Cv_kin = (3 * n_atoms * k_B / 2) / (1  - 2 / (3 * n_atoms * k_B**2 * T_avg**2) *
    ↪  dE_kin) * e
    Cv_pot = (3 * n_atoms * k_B / 2) / (1  - 2 / (3 * n_atoms * k_B**2 * T_avg**2) *
    ↪  dE_pot) * e

    return Cv_kin, Cv_pot, T_avg, dE_kin, dE_pot

data_3, _, _, _, its_eq_3, _, _, _ = read_data(0.001, 40000, 25000, task=3)
data_4, _, _, _, its_eq_4, _, _, _ = read_data(0.001, 50000, 30000, task=4)

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)
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)

results = pd.DataFrame({'Cv_kin': [Cv_kin_3, Cv_kin_4],
                       'Cv_pot': [Cv_pot_3, Cv_pot_4],
                       'dE_kin': [dE_kin_3, dE_kin_4],
                       'dE_pot': [dE_pot_3, dE_pot_4],
                       'T_avg': [T_avg_3, T_avg_4]},
                       index=['Solid', 'Liquid'], )
results = results.style.format({"Cv_kin": "{:.3e}", "Cv_pot": "{:.3e}",
                               "dE_kin": "{:.3f}", "dE_pot": "{:.3f}", "T_avg": "{:.3f}"})
display(results)

######################################## Task 6 ########################################
def plot_rdf(r, rdist, r_m, r_m_ind, n, save=False):
    fig, ax = plt.subplots(figsize=(8, 6))
    plt.plot(r, rdist, 'k', label='RDF', lw=2.5)
    plt.axvline(r_m, ymin=0, ymax=0.2, color='k', linestyle='--')
    plt.plot(r[r_m_ind], rdist[r_m_ind], '*', color='tab:orange', markersize=15,
    ↪  label='$r_m$')
    plt.fill_between(r[:r_m_ind+1], rdist[:r_m_ind+1], alpha=0.5, color='k')
    plt.ylim(0, 3)
    plt.xlabel('$r$ [Å]')
    plt.ylabel('$g(r)$')
    plt.legend(loc='upper right')
    plt.tight_layout()

    return fig

```

```python
save = False
rdist_data, its, t_max, delta_t, its_eq, N_bins, a_0, _ = read_data(0.001, 50000, 30000,
↪  6)
rdist = np.cumsum(rdist_data, axis=0)[-1, :]/len(rdist_data)

r = np.linspace(0, a_0*2, N_bins)
r_m_ind = np.argmin(rdist[50:]) + 50
r_m = r[r_m_ind]
n = n_atoms / np.power(4*a_0, 3)

I_r = 4 * n * np.pi * np.trapezoid(rdist[:r_m_ind] * r[:r_m_ind]**2, r[:r_m_ind])
print(f'The coordination number is {I_r:.3f}')

fig = plot_rdf(r, rdist, r_m, r_m_ind, n)

if save:
    save_fig(fig, 'rdf', 6)

######################################## Task 7 ########################################
def plot_sfact(q, sfact):
    fig, ax = plt.subplots(figsize=(8, 6))

    plt.plot(q, sfact, 'k', label='Structure Factor', lw=2.5)

    plt.xlabel('$q$ [Å$^{-1}$]')
    plt.ylabel('$S(q)$')
    plt.legend(loc='upper right')
    plt.tight_layout()

    return fig

save = False
sfact_data, its, _, delta_t, its_eq, N_bins, a_0, N_max = read_data(0.001, 50000, 30000,
↪  7, opt='big')
sfact = np.cumsum(sfact_data, axis=0)[-1, 1:]/len(sfact_data)

q_max = 2 * np.pi / (4*4.25) * N_max * np.sqrt(3)
q = np.linspace(0, q_max, int(N_bins))
print(f'The first peak of the structure factor is at {q[np.argmax(sfact)]:.3f}
↪  Å$^{{-1}}$')

fig = plot_sfact(q, sfact)

if save:
    save_fig(fig, 'sfact', 7)
```