H1a: MD-simulation of aluminium properties

Carl Strandby and Didrik Palmqvist

November 2022

Introduction

A physicist studying matter interactions on a macroscopic scale will often encounter problems that either can't be solved analytically, or where it is not practical to do so. In such cases the physicist must resort to numerical methods. One such method, molecular dynamics simulation, or MD-simulations for short is a technique where interacting particle trajectories are generated by numerically integrating Newton's equation of motion [1]. This paper examines static properties of aluminium undergoing phase changes, by using MD-simulation together with the velocity-verlet-algorithm for integrating particle trajectories. Sections () to () outlines different problems and simulations, together with their theoretical background and analysis of the simulation results. The last section, Section , contains a discussion of the respective results, and the advantages and disadvantages of the MD-simulation technique. The purpose of this paper is to theoretically outline, and apply a basic MD-simulation technique which can later be generalized to study more complex molecular and lattice behaviours.

Problem 1

Solid aluminium, Al, organizes as a face-centered-cubic-crystal, or fcc-crystal. The length between two opposing sides of the fcc-crystal is called the lattice-parameter. The total binding energy of the fcc-crystal depends on the aluminium particles inter-atomic potentials, which in turn is a function of the aforementioned lattice parameter. One can vary the lattice parameter and examine how this changes the total binding energy. The value of the lattice parameter that minimizes the total binding energy is then the lattice parameter that is found in naturally occurring aluminium. The first problem in this paper is to identify this equilibrium lattice parameter numerically.

The first step is to initialize an FCC-structure. This was done using a function included in the course material, which can be found in appendix (C.2). The created FCC-structure is a supercell of four unit cells in each direction. The number of atoms for one unit cell is one eighth per corner, and one half per face, for a total of four atoms. Hence the supercell contains $4 \times 4^3 = 256$ atoms. The reason for using a supercell instead of just examining one unit cell, is to capture effects from particle interactions over distances longer than the lattice parameter. But these are usually weak over distances longer than a few unit cells and thus interactions between particles longer is approximately zero. Assuming that a particle in a homogenous material has a spherical range of interactions, the particles on the sides of a unit cell should interact with particles in neighbouring cells. These effects can be taken into account by using the supercell and applying periodic boundary conditions. Figuratively, the supercell can be seen as periodically repeating "images" of the unit cell, see figure (1), and implemented using the minimum image convention [2]. Results can then be calculated from considering a particles interaction with the nearest images of surrounding particles, which is then averaged and reported in units of per unit-cell.

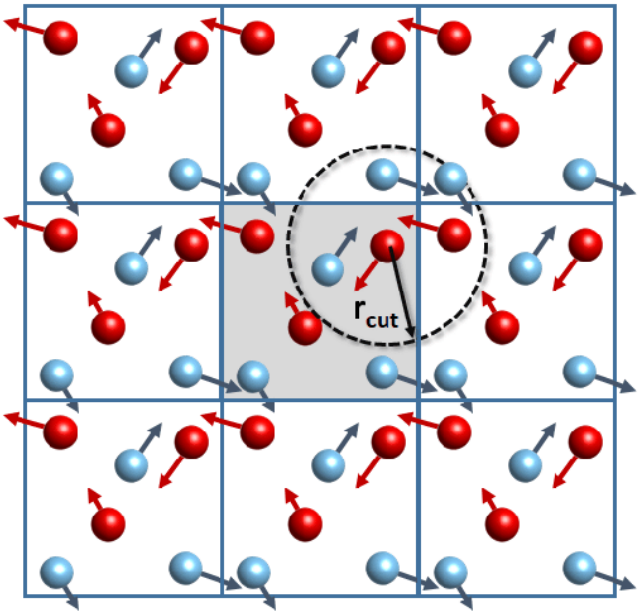


Figure 1: Periodic boundary conditions used to model interactions between particles within eachothers interaction range but placed in different unit cells. In this problem, a supercell of $4 \times 4 \times 4$ was used. Picture by Bo Yang [3]

The FCC-cell was initialized with 20 lattice parameters uniformly spaced out between 3.98 and 4.075 and their respective potential energies where calculated. The equilibrium lattice parameter was determined to be (4.030 ± 0.005) Å and the resulting plot is seen in Figure (2). A slight jump in energy for the measured datapoints can be seen around value of the lattice par for the local minimum. One possible cause could be the use of the minimum image convention, but it is uncertain.

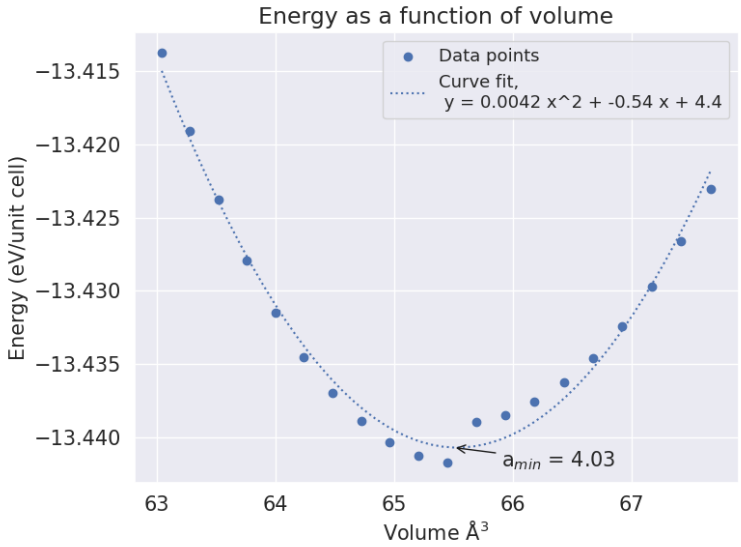


Figure 2: Potential energy of an aluminium fcc as a function of the lattice parameter. The local minimum is the equilibrium lattice parameter, here identified as (4.030 ± 0.005) Å. A slight jump in measured energy of measured datapoints can be seen around this value of the lattice parameter.

Problem 2

A common algorithm to perform the numerical integration of particle trajectories is the Velocity-Verlet-algorithm. The algorithm is derived from taylor expanding the position coordinate forward and backwards in time, and is summarized in the equation relating the velocity in a following timestep to the acceleration as

$$\mathbf{v}(t + dt) = \mathbf{v}(t) + \frac{1}{2}[\mathbf{a}(t) + \mathbf{a}(t + dt)]dt. \tag{1}$$

[2]. A clever implementation of the algorithm, made to reduce the error, divides this timestep into two parts. In the first half the contribution of the first acceleration term, $\mathbf{a}(t)$, is used to update the velocity at $\mathbf{v}(t + dt/2)$. This velocity is then used to update the particle position, r, after which a new acceleration, $\mathbf{a}(t + dt)$, is retrieved, and used to update the velocity at $\mathbf{v}(t + dt)$. This can be written in pseudocode as

$$\begin{aligned}
&\mathbf{v}(t + dt/2) = \mathbf{v}(t) + \tfrac{1}{2}\mathbf{a}(t)dt \\
&\mathbf{r}(t + dt) = \mathbf{r}(t) + \mathbf{v}(t + dt/2)dt \\
&\mathbf{a}(t) \rightarrow \mathbf{a}(t + dt) \\
&\mathbf{v}(t + dt) = \mathbf{v}(t + dt/2) + \tfrac{1}{2}\mathbf{a}(t + dt)dt
\end{aligned} \tag{2}$$

The benefits of the Velocity-Verlet-algorithm is that it is time-reversible, has a small error-term, $O((\Delta)^3)$, and is self-starting, meaning that the state in the next time-step only depends on the state in the previous [4]. But it also assumes that the forces acting on the particles only depend on their position and not their velocity. This means that the algorithm has to be adapted to calculate trajectories for charged particles in an electromagnetic field for example.

In this problem the particles of the equilibrated aluminium FCC-cell is displaced and their trajectories, numerically integrated with the Velocity-Verlet-algorithm, is examined to calculate the temperature and pressure per unit-cell. The particles is first displaced with a factor of ±6.5% of the lattice-parameter. The forces on each particle was retrieved with a function provided in the course material, see appendix (C.1), after which followed a sequence of timesteps. One timestep consisted of first calculating the velocities half a timestep forward, then the positions after which forces where updated and used to update the velocities another half timestep forward, see equation (2). The velocities after the timestep was used to calculate the particles kinetic energy as

$$E_{\text{kin}} = \left\langle \sum_{i=1}^{N} \frac{p_i^2(t)}{2m_i} \right\rangle_t. \tag{3}$$

And from the kinetic energy the temperature, T, is calculated using the equipartition theorem,

$$T = \frac{2}{3Nk} E_{\text{kin}}, \tag{4}$$

where N is the total number of particles and k is Boltzmanns constant. The pressure, P, is in turn given by

$$PV = NkT + W, \tag{5}$$

where W is called the virial, which was retrieved with a provided function and accounts for some intra-molecular forces, see appendix (C.1) [5]. The virial is given by

$$W = \langle \mathcal{W} \rangle_{NVT} = \left\langle \frac{1}{3} \sum_{i=1}^{N} \boldsymbol{r}_i \cdot \boldsymbol{F}_i \right\rangle_{NVT}. \tag{6}$$

Lastly the potential energy was retrieved with a provided function, from which the total energy is simply given by the sum of the kinetic and potential energy. The full source code of the implementation can be seen in Appendix (A.3).

One of the parameters for the algorithm is the size of the timestep, dt. An optimal timestep will be small enough that energy will be conserved during simulation, but also big enough to not unnecessarily prolong simulation time. Figures (3) to (3) shows a timeseries over measured energy for different value of dt. The rightmost graph showing total energy can be seen to show a relative conservation for dt equals to $1e-2$, complete divergence for $2e-2$ and a slight drift for $1.5e-2$ which could indicate an imminent divergence. Hence the value of $1e-2$ was chosen as an upper limit for dt.

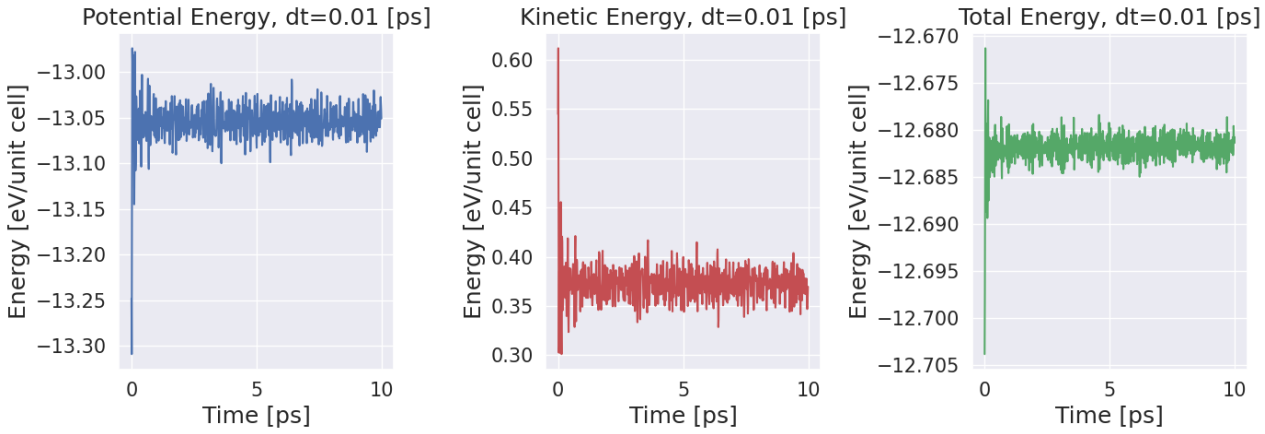


Figure 3: Energy over time for an aluminium fcc lattice. In the graph to the right, the total energy can be seen to be relatively conserved.

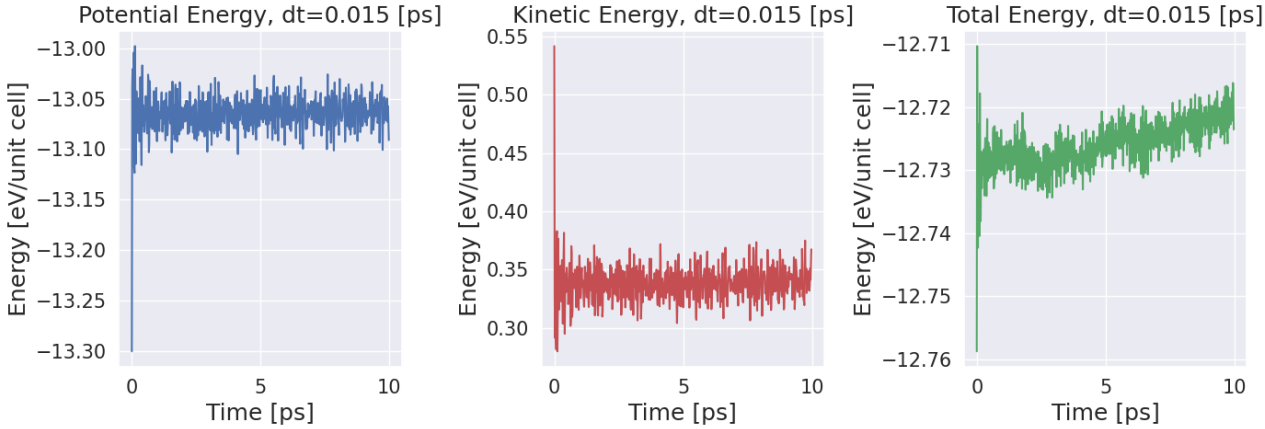


Figure 4: Energy over time for an aluminium FCC lattice. In the graph to the right, the total energy can be seen to be drifting, which could indicate an imminent divergence.

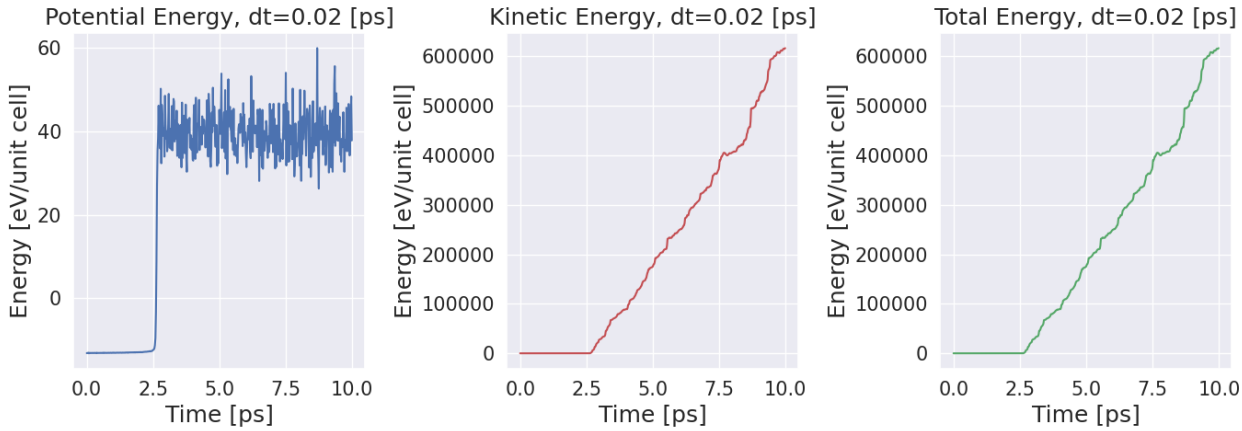


Figure 5: Energy over time for an aluminium FCC lattice. In the graph to the right, conservation of the total energy is violated.

The temperature per unit cell of the simulation with $dt = 1e-2$ can be seen in figure (6), where it shows fluctuating behaviour around 720.93 K. It can be noted that the fluctuation in the temperature and energy graphs is natural as an initial displacement in a frictionless environment should lead to the particle-oscillations around their equilibrium-position. The magnitude of the average temperature is affected by the size of the random displacements, here ±6.5% of the lattice parameter, with a larger displacement leading to a higher average temperature as the oscillations would be larger. The higher temperatures in the first few timesteps is natural as the particles because all particles start away from their equilibrium and need a few timesteps of interaction before they occupy a more stable state. Note also that here quantities are measured at the end of the velocity verlet timestep, and therefore kinetic energy and temperature doesn't start from zero.

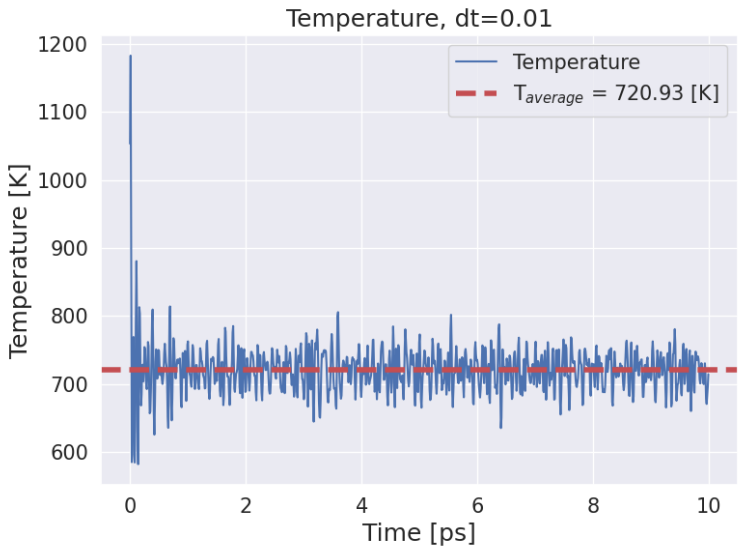


Figure 6: Temperature over time for an aluminium FCC lattice. The temperature can be seen to fluctuate around 720.93 K.

Problem 3

When performing a molecular-dynamics simulation we often are interested in studying a system with certain macroscopic properties such as temperature or pressure. When specifying initial conditions for the particles being simulated and equilibrating the system it will tend to a certain temperature and pressure, but we are not able to obtain a specific macroscopic state by choosing certain initial positions and velocities for the particles. For this problem we where tasked with simulating aluminium in a solid phase at the temperature 500°C (773.15 K) and pressure 1 Bar. A common technique to circumvent this problem is to scale coordinates for position and velocites during the equilibration after which a production run is performed from which average quantities can be calculated. For this project the following algorithm was used in combination with velocity verlet for scaling the velocity coordinates

$$\mathbf{v}_i^{\text{new}} = \sqrt{\alpha_T}\mathbf{v}_i^{\text{old}} \tag{7}$$

with

$$\alpha_T = 1 + \frac{2dt}{\tau_T}\frac{T_{\text{eq}} - \mathcal{T}(t)}{\mathcal{T}(t)} \tag{8}$$

where $\mathcal{T}(t)$ is the instanteneous temperature, τ_T is a time constant regulating the speed with which the temperature is re-scaled, dt is the size of the time step and T_{eq} is the equilibrium temperature. The position coordinates and the lattice parameter a_0 where re scaled by

$$\mathbf{r}_i^{\text{new}} = {\alpha_P}^{1/3}\mathbf{r}_i^{\text{old}}, \quad {a_0}^{\text{new}} = {\alpha_p}^{1/3}{a_0}^{\text{old}} \tag{9}$$

with

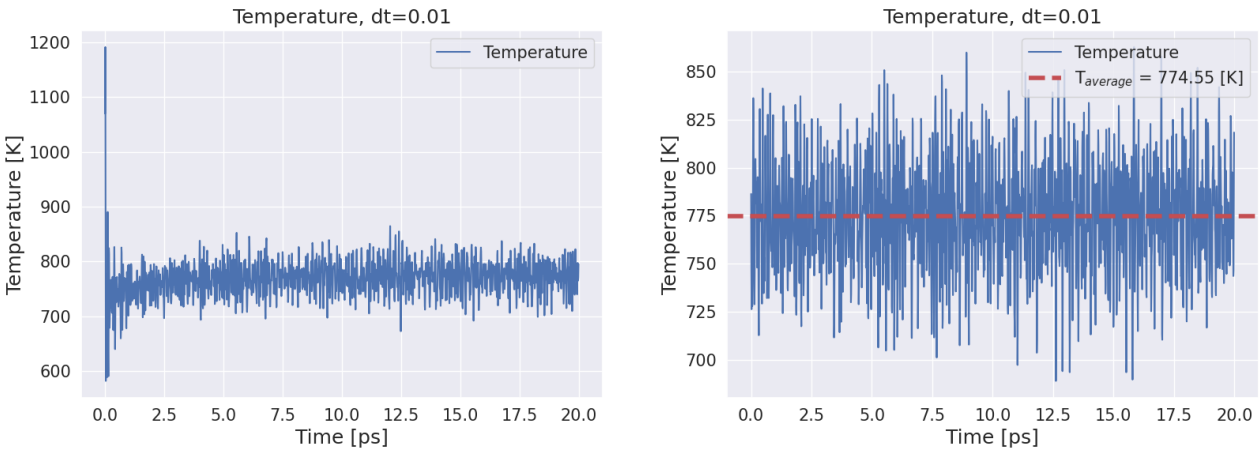
$$\alpha_P = 1 - \kappa_T \frac{dt}{\tau_P} \frac{P_{\text{eq}} - \mathcal{P}(t)}{\mathcal{P}(t)} \tag{10}$$

where $\mathcal{P}(t)$ is the instantenous pressure, P_{eq} is the equilibrium pressure, τ_P is a time constant regulating the speed of the pressure scaling. The κ_T is the isothermal compressibility, being $0.013\,85\,\text{Bar}^{-1}$ for aluminium [6]. The size of parameters τ_T and τ_P were chosen to be $100 \cdot dt$ and $300 \cdot dt$ by trial and error according to what yielded good results in terms of liquidization and energy conservation of the system. A summary of the parameters used for simulation of solid aluminium can be seen in table (1).

The resulting graphs of temperature and pressure during equilibration and production can be seen in figures (11) and (12) respectively. The time average of the temperature for the production run was calculated to be $\langle T \rangle = 774.55\,\text{K}$, with fluctuations of up to 100 K.

dt [ps]	κ_T [1/Bar]	T_{eq} [K]	P_{eq} [Bar]	τ_T [ps]	τ_P [ps]
0.01	$0.01385 \cdot 10^4$	773.15	1	$100 \cdot dt$	$300 \cdot dt$

Table 1: Table presenting parameter values used for simulation of solid aluminium. Value for κ_T was retrieved from [6] and then converted to inverse bar.

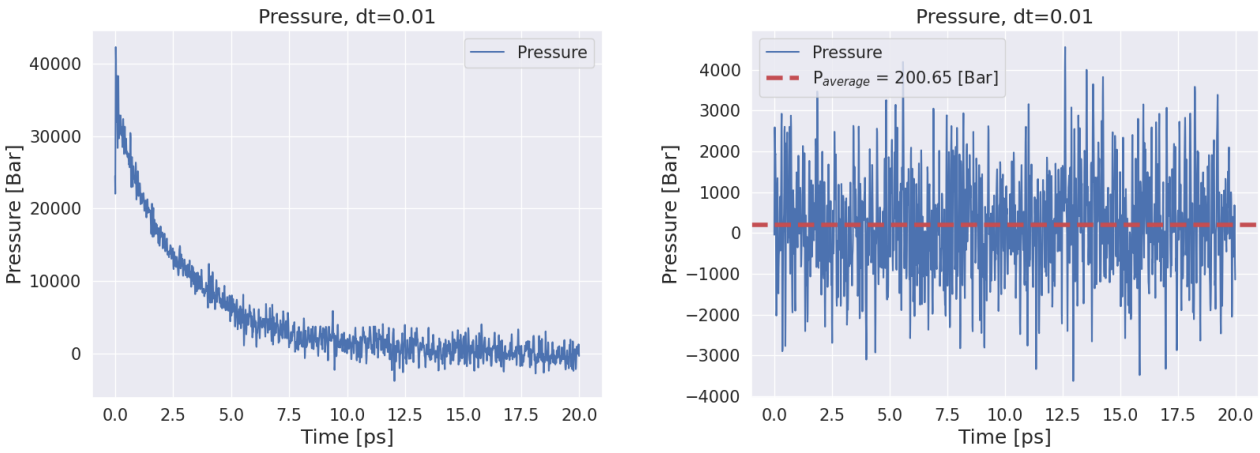


(a) Temperature during equilibration.

(b) Temperature during following production run.

Figure 7: Equilibration and production temperature during simulation of solid aluminium.

Pressure as a function of time can be seen in figure (8a) and (8b). The time average of the pressure for the production run was calculated to be $\langle P \rangle = 200.65\,\text{Bar}$, a bit larger than the desired pressure of 1 Bar. One reason could be that the temperature is a macroscopic quantity, that is not very well defined for smaller systems, such as this one with only 256 particles. One should also keep in mind that this computer simulated system doesn't take all possible particle interactions into account, and won't give an entirely accurate representation of a real physical system.



(a) Pressure during equilibration.

(b) Pressure during production run.

Figure 8: Equilibration and production pressure during simulation of solid aluminium. Pressure during production can be seen fluctuating heavily.

One consequence of the large fluctuations of pressure is that it is difficult to see whether the system is fully equilibrated. Therefore one can look at the lattice parameter that is scaled with α_P, see equation 9, and define an adequate pressure equilibration as one in which the lattice parameter has stabilized. The measurement of the lattice parameter for the solid aluminium simulation is shown in figure (9). There the lattice parameter can be seen stabilize and become appriximately constant around 17.5 ps, which can be compared to pressude during equilibration also stabilizing, see figure (8a). The final equilibrated lattice parameter used for the production run was 4.092 Å.

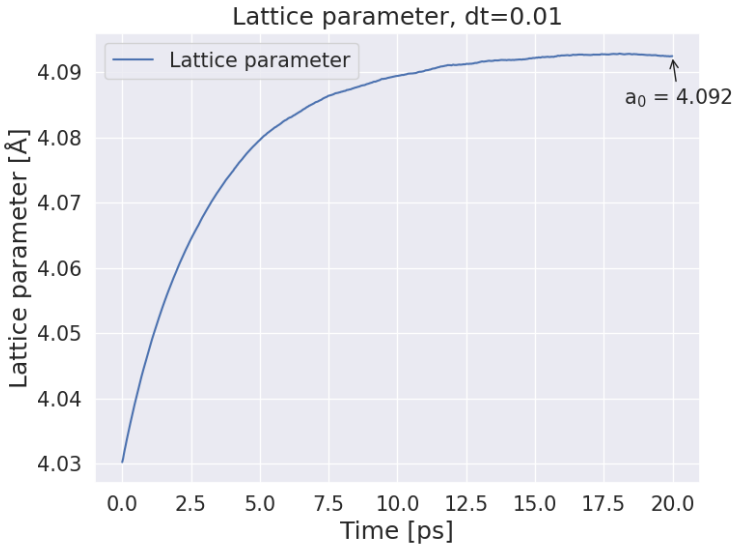


Figure 9: Lattice parameter as a function of time during equilibration. The last measured value of 4.092 Å was used for the production run.

Lastly, to show that the system was in a solid state the trajectories of three random particles were plotted in figure (10). These plots clearly shows the particles remaining close to their initial positions.

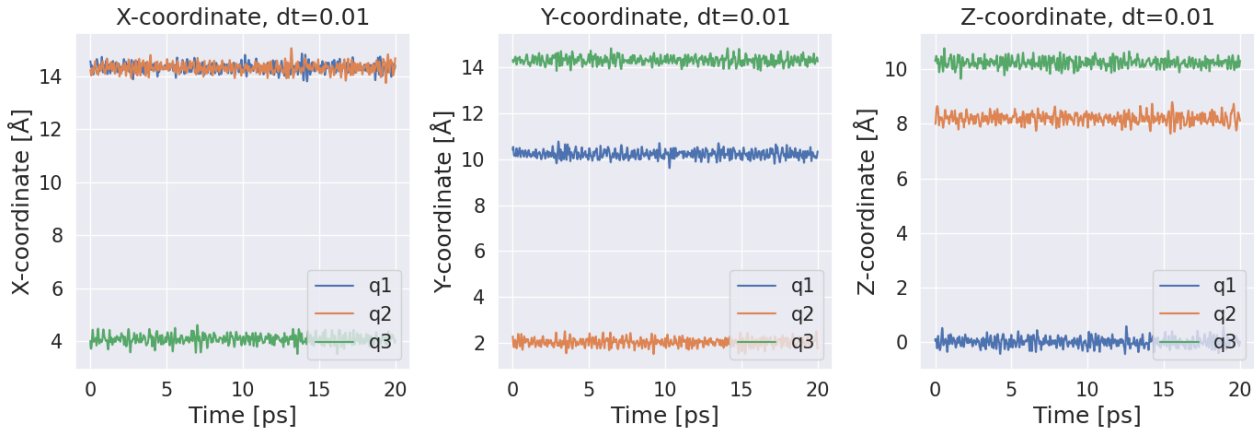
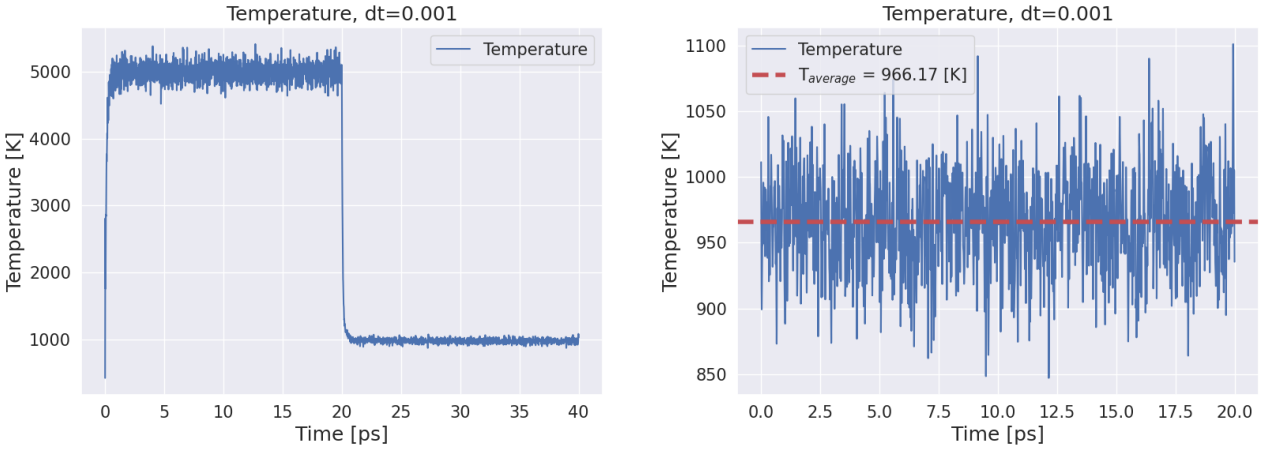


Figure 10: Positions of three random particles during production run of solid aluminium.

Problem 4

Simulation of aluminum in a liquid phase, at a temperature of 973.15 K and pressure of 1 Bar, was implemented using a similar equilibration procedure to the one discussed in the previous section, but with an extra equilibration step. The first equilibration step was performed at a higher temperature of 5000 K for 20 ps after which an equilibration run was performed at 973.15 K for another 20 ps. The reason for heating up the system to a temperature above the desired, in contrast to how a real-world procedure would be conducted, is that in the real world materials often need a nucleation point to start melting. And this is not present in the idealised version realized in a simulation. Also, the timestep was changed from 10^{-2} ps to 10^{-3} ps to ensure that the system was well behaved as the increased dynamics of the system needed the smaller timestep for energy to be conserved during the production run. After these initial steps of heating up and cooling down the system, a production run was performed for 20 ps where time averages of the pressure and temperature could be calculated. The time average of the temperature was calculated to $\langle T \rangle = 966.17$ K and the time dependence of the temperature is displayed in figures (11a) and (11b) for the equilibration and production steps respectively.



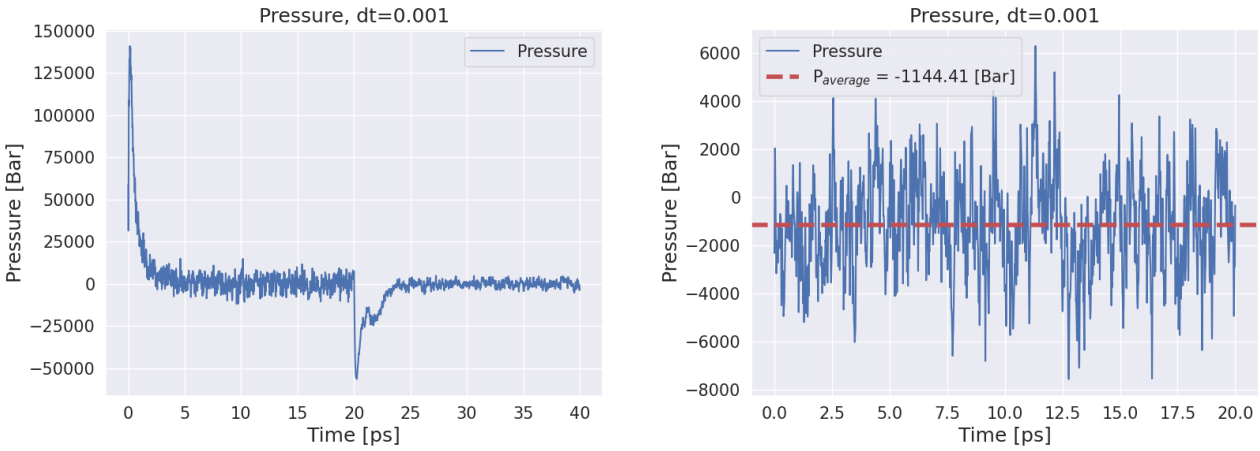
(a) Temperature during equilibration

(b) Temperature during production run.

Figure 11: Equilibration and production temperature during simulation of liquid aluminium. During equilibration temperature is first increased heavily, before a cool-down period is initiated, where the system is equilibrated against the desired temperature of 973.15 K. In the following production run, the average temperature was measured to be 966.17 K.

The time average of the pressure for the liquid aluminium simulation was calculated during the production run to be

$\langle P \rangle = -1144.41$ Bar and the pressure as a function of time can be seen for the equilibration and the production run in figures (12a) and (12b). The corresponding measurement for the lattice parameter can be seen in figure 13, where the lattice parameter can be seen to be stabilized at the end the heating step at 20 ps, and again at the end of the cooling step at 40 ps. The final lattice parameter used for the production run was 4.254 Å, slightly bigger than the lattice parameter of solid aluminium simulation of 4.092 Å. One should note that a liquid doesn't have a crystal lattice structure, and thus the concept of a lattice parameter is not well defined. But it could be argued that the lattice parameter in this simulated system indicates that the atoms in the equilibrated state are spaced further apart, than in the inital state. Which is true for most newtonian solids and fluids when temperature is increased.



(a) Pressure during equilibration

(b) Pressure during production run.

Figure 12: Equilibration and production pressure during simulation of liquid aluminium. During equilibration temperature is first increased heavily, before a cool-down period is initiated, where the system is equilibrated against the desired pressure of 1 Bar. In the following production run, the average pressure was measured to be −1144 Bar and can be seen to be heavily fluctuating.

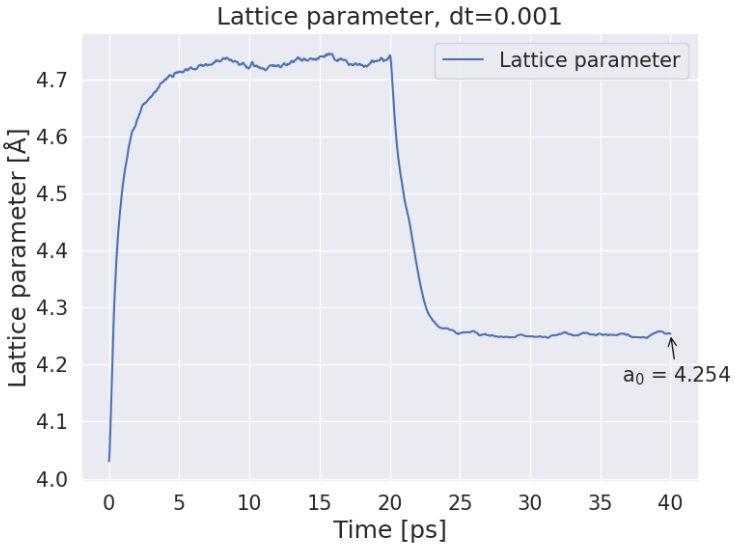


Figure 13: Lattice parameter as a function of time during equilibration of a liquid aluminium simulation. The last measured value of 4.254 Å was used for production run.

To determine that the system had indeed entered a liquid phase the trajectories of a few random particles during the

production run were plotted in figure 14. There it can be seen that the particles travel a distance of a few to a few dozen Ångström from their initial positions in a few picoseconds. In contrast to the more constant positions seen in the solid aluminium simulation in figure 10, the particles could be said to be in a liquid state.

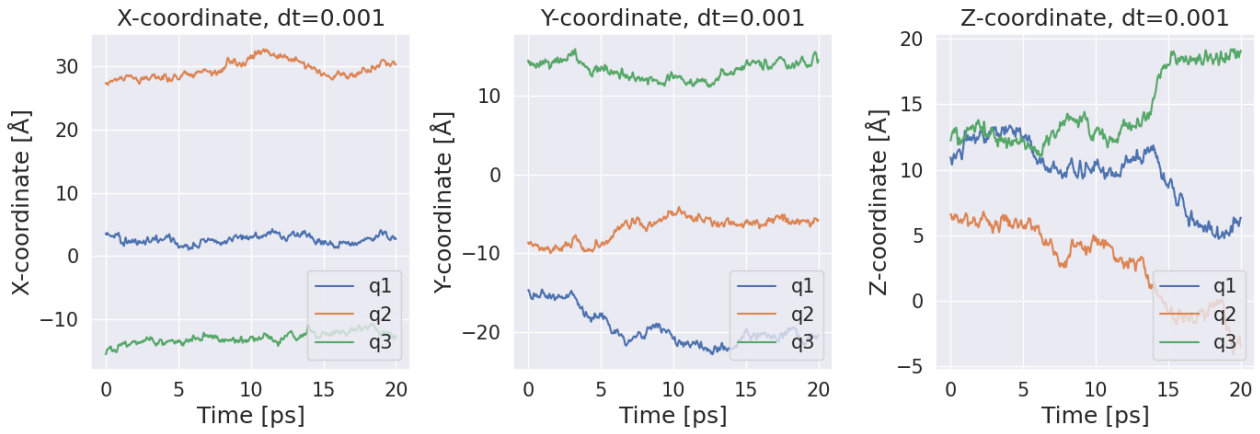


Figure 14: Positions of three random particles during production run of solid aluminium.

Problem 5

For a system with constant number of particles N, constant volume V and constant energy E the heat capacity at constant volume C_V is related to the fluctuations of the instantaneous values of the potential and kinetic values. The heat capacity can be calculated as

$$
\begin{aligned}
C_{V,pot} &= \frac{3Nk_B}{2}\left[1 - \frac{2}{3Nk_B^2T^2}\langle(\delta\varepsilon_{pot})^2\rangle_{NVE}\right]^{-1} \\
C_{V,kin} &= \frac{3Nk_B}{2}\left[1 - \frac{2}{3Nk_B^2T^2}\langle(\delta\varepsilon_{kin})^2\rangle_{NVE}\right]^{-1}
\end{aligned}
\tag{11}
$$

where $\langle(\delta\varepsilon_{pot})^2\rangle_{NVE}$ and $\langle(\delta\varepsilon_{kin})^2\rangle_{NVE}$ are the variance of the instantaneous values of the potential and kinetic energy along the phase-space trajectory of the simulations. Values for the heat capacity were calculated for solid aluminum at $T = 500°$ C at $P = 1$,bar and liquid aluminum at $T = 700°$ K and $P = 1$ bar. The values calculated from the simulations are presented in table (2). The values calculated using fluctuations in kinetic and potential energy are close which would be expected for a system with conserved total energy as a fluctuation in kinetic energy would imply a fluctuation in potential energy and vice versa. One can also notice that the values for the heat capacity for the solid and melted system also are very close, which is not expected as the experimental value of the specific heat capacity for solid aluminum is 921.096 J/(kg K) [7] and 1.18J/(kg K) [8]. The simulated values are a lot closer to the value the experimental value of aluminum, this implies that there are some problems simulating the liquid state of aluminum in a way that is true to the real physical system.

	$C_{V,pot}$ [eV/K]	$C_{V,kin}$ [eV/K]
$T = 500°C$, $P =$ bar	0.0661900	0.0661893
$T = 700°C$, $P = 1$ bar	0.0661895	0.0661895

Table 2: Value for the heat capacity calculated from fluctuations in the potential and kinetic energies during the simulation for a solid and liquid state for aluminium.

Problem 6

For this problem we were tasked with determining the radial distribution function $g(r)$ for the aluminium in a liquid state with the temperature $T = 700°$C and pressure $P = 1$ bar. The pair distribution function for a spatially homogeneous system

is given by

$$g(\mathbf{r}', \mathbf{r}'') = \left\langle \sum_{i=1}^{N} \sum_{j \neq i}^{N} \delta(\mathbf{r}' - \mathbf{r}_i)\delta(\mathbf{r}'' - \mathbf{r}_j) \right\rangle \tag{12}$$

and for a isotropic system we can average over angles and not lose information and thus calculate the radial distribution function for the system. By integrating the radial distribution function from zero to its first minimum r_m we obtain the coordination number

$$I(r_m) = n \int_0^{r_m} g(r) 4\pi r^2 dr, \tag{13}$$

where n is the density of particles. The calculation of $g(r)$ was implemented in the Velocity-Verlet where at each time step the distance between every pair of particles were calculated. To handle the periodic boundary conditions the minimum image convention was used and the distances were sorted into a histogram and then by averaging over all particles and each time step a value for $g(r)$. The histogram was constructed by creating bins according to

$$r_k = (k - 0.5)\Delta r, \; k = 1, 2, 3, \; ... \, 300. \tag{14}$$

What bin to count a recorded distance r_{ij} between two particles was determined by rounding down the expression $r_{ij}/(\Delta r) + 0.5$ to the closest integer and adding a count to the an array at the corresponding index. The resulting value for the radial distribution function can be seen in figure (15). The coordination number was calculated to $I(r_m) = 12.521$ using `scipy.integrate.trapezoid()` in python from the measured minimum of 4.22 Å, see figure (15). The resulting value is relatively close to the value of solid aluminiums coordination number 12 and according to [9] a value of 11.5 was obtained for liquid aluminum so this result isn't unreasonable.

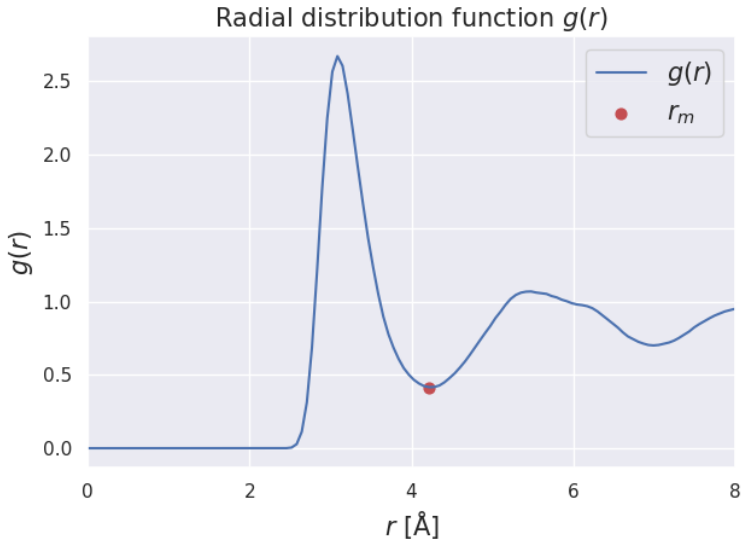


Figure 15: Figure desplaying the radial distribution function obtained from simulating aluminum at $T = 700°C$ and $P = 1$ bar. First local minium used to calculate coordination number was 4.22 Å.

Concluding Discussion

In general the results seen indicates the utility of computer simulations using the velocity verlet algorithm. Using the produced and provided code, quantities such as the lattice parameter, temperature, pressure, heat capacity and coordination number has been examined for solid and liquid aluminium respectively. The simulated values of the lattice parameter for solid aluminium at 773.15 K and liquid aluminium at 973.15 K has been measured to 4.092 and 4.254 Å respectively. These can be compared to experimental values of 4.0478 Å [10], which is relatively close. However, the lattice parameter for a

liquid is not well defined, and it is thus interesting that such a reasonable result was obtained. One reason could be the short timescale used in the simulation, and so the lattice parameter for the liquid aluminium simulation could be predicted to diverge during a longer simulation. While measured temperatures and pressures were consequences of equilibration, it was interesting to note the heavy fluctuations of pressure, compared to the relatively more well behaved fluctuations of temperature. This was discussed to partly be a consequence of the small system of just 256 particles, but this was not examined further and would be an interesting continuation of the experiment. The measured heat capacity for the solid and liquid aluminium simulations where measured to be very similar, which was not expected as the experimental values for these differ significantly. Lastly the coordination number was measured to be 12.521 which was relatively close to experimental values [9].

These results indicates the advantages of molecular dynamics simulations using the velocity verlet algorithm, as it allows the comparison between theoretical results that otherwise couldn't be obtained with great accuracy, to experimental results. One could also use results obtained to identify which parameters are more likely to impact the outcomes and use that to design experiments. For example one could use the scale of interactions to guide the selection of accuracy of measurement devices. In conclusion one could say that the results shown in this paper shows the utility of molecular dynamics simulations with the velocity verlet algorithm.

References

[1] Kunal Roy, Supratik Kar, and Rudra Narayan Das. Chapter 5 - computational chemistry. In Kunal Roy, Supratik Kar, and Rudra Narayan Das, editors, *Understanding the Basics of QSAR for Applications in Pharmaceutical Sciences and Risk Assessment*, pages 151–189. Academic Press, Boston, 2015.

[2] Göran Wahnström. Md lecture notes, Oct 2021.

[3] Bo Yang. *Mechanical properties of graphene-borophene heterostructures grain boundaries Molecular dynamic modeling*. PhD thesis, 09 2020.

[4] Q Spreiter and M Walter. Classical molecular dynamics simulation with the velocity verlet algorithm at strong external magnetic fields. *Journal of Computational Physics*, 152(1):102–119, 1999.

[5] Yaşar Demirel and Vincent Gerbaud. Chapter 1 - fundamentals of equilibrium thermodynamics. In Yaşar Demirel and Vincent Gerbaud, editors, *Nonequilibrium Thermodynamics (Fourth Edition)*, pages 1–85. Elsevier, fourth edition edition, 2019.

[6] Charles Kittel. *Introduction to Solid State Physics*. John Wiley & Sons, Inc, Hoboken, NJ, 2005.

[7] Metals - specific heats, 2003. Accessed 9 12 2022.

[8] Metals - boiling points and specific heat, 2014. Accessed 09 12 2022.

[9] Dong-Ping Tao. Prediction of the coordination numbers of liquid metals. *Metallurgical and Materials Transactions A*, 36:3495–3497, 12 2005.

[10] Masahiko Morinaga. 6 - aluminum alloys and magnesium alloys. In Masahiko Morinaga, editor, *A Quantum Approach to Alloy Design*, Materials Today, pages 95–130. Elsevier, 2019.

A Source Code

A.1 Running program and setting parameters for MD-simulations: `run.c`

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <stdbool.h>

#include "lattice.h"
#include "potential.h"
#include "tools.h"
#include "try_lattice_constants.h"
#include "velocity_verlet.h"

#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <time.h>

void H1_task1(), H1_task2(), H1_task3(), H1_task4(), H1_task6();

int
run(
    int argc,
    char *argv[]
   )
{
    H1_task1();
    H1_task2();
    H1_task3();
    H1_task4();
    H1_task6();

    return 0;
}

void
H1_task1()
{
    int nbr_atoms = 256; int n_rows = nbr_atoms; int n_cols = 3; int n_unitcells = 4;

    // Testing lattice parametersdhwudh
    bool get_small_lattice_param = true;
    if(get_small_lattice_param == true){
    double smallest_lattice_param = 0;
    smallest_lattice_param = try_lattice_constants((int) nbr_atoms, (int) n_rows, (int) n_cols);
    printf("%f\n", smallest_lattice_param);
    }
}

void
H1_task2()
{
     int nbr_atoms = 256; int n_rows = nbr_atoms; int n_cols = 3; int n_unitcells = 4;

    // Initialising position and velocity arrrays
    double position[nbr_atoms][n_cols];
    double velocity[nbr_atoms][n_cols];
    for(int ix = 0; ix < nbr_atoms; ix++){
        for(int jx = 0; jx < n_cols; jx++){
            velocity[ix][jx] = 0;
        }
    }

    // Choosing lattice param
    double lattice_param = 4.03; // True is around 4.0478    (Masahiko Morinaga, https://bit.ly/3ERRFt3)
    double cell_length = 4 * lattice_param;

    // Initialice and displace fcc
    init_fcc((double (*)[3]) position, (int) n_unitcells, (double) lattice_param); // 4 unit cells in each ↩
        direction

    displace_fcc((double (*)[3]) position, (int) n_unitcells, (double) lattice_param);

    // Declaring parameters for velocity verlet.
    // If temp/press_scaling = false scaling is turned off and scaling factors remains = 1
    int end_time; double dt;
    bool temp_scaling, press_scaling, write_not_append;
    double temp_eq, press_eq, tau_T, tau_P;

```

```
76      // Production run
77      end_time = 10; dt = 2e-2;
78      temp_scaling = false, press_scaling = false;
79      temp_eq = 773.15; press_eq = 1; //773.15 K och 1 Bar
80      tau_T = 1 * dt; tau_P = 1*dt;
81      write_not_append = true;
82
83      cell_length = velocity_verlet((double (*)[3]) position, (double (*)[3]) velocity, (double) lattice_param, (↩
            double) cell_length, (int) end_time, (double) dt, (int) n_cols, (int) nbr_atoms, \
84                      (bool) temp_scaling, (bool) press_scaling, (double) temp_eq, (double) press_eq, (bool) ↩
                            write_not_append, (double) tau_P, (double) tau_T);
85  }
86
87  void
88  H1_task3()
89  {
90      int nbr_atoms = 256; int n_rows = nbr_atoms; int n_cols = 3; int n_unitcells = 4;
91
92      // Initialising position and velocity arrrays
93      double position[nbr_atoms][n_cols];
94      double velocity[nbr_atoms][n_cols];
95      for(int ix = 0; ix < nbr_atoms; ix++){
96          for(int jx = 0; jx < n_cols; jx++){
97              velocity[ix][jx] = 0;
98          }
99      }
100
101     // Choosing lattice param
102     double lattice_param = 4.03; // True is around 4.0478    (Masahiko Morinaga, https://bit.ly/3ERRFt3)
103     double cell_length = 4 * lattice_param;
104
105     // Initialice and displace fcc
106     init_fcc((double (*)[3]) position, (int) n_unitcells, (double) lattice_param); // 4 unit cells in each ↩
            direction
107
108     displace_fcc((double (*)[3]) position, (int) n_unitcells, (double) lattice_param);
109
110     // Declaring parameters for velocity verlet.
111     // If temp/press_scaling = false scaling is turned off and scaling factors remains = 1
112     int end_time; double dt;
113     bool temp_scaling, press_scaling, write_not_append;
114     double temp_eq, press_eq, tau_T, tau_P;
115
116     // end_time = 25; dt = 1e-3;
117     // tau_T = 100*dt; tau_P = 50*dt;
118
119     // Equalibration run
120     end_time = 20; dt = 1e-2;
121     temp_scaling = true; press_scaling = true;
122     temp_eq = 773.15; press_eq = 1; //773.15 K och 1 Bar
123     tau_T = 100*dt; tau_P = 300*dt; //*dt; 50,5
124     write_not_append = true;
125
126     cell_length = velocity_verlet((double (*)[3]) position, (double (*)[3]) velocity, (double) lattice_param, (↩
            double) cell_length, (int) end_time, (double) dt, (int) n_cols, (int) nbr_atoms, \
127                     (bool) temp_scaling, (bool) press_scaling, (double) temp_eq, (double) press_eq, (bool) ↩
                            write_not_append, (double) tau_P, (double) tau_T);
128
129
130     // Production run
131     end_time = 20; dt = 1e-2;
132     temp_scaling = false, press_scaling = false;
133     temp_eq = 773.15; press_eq = 1; //773.15 K och 1 Bar
134     write_not_append = true;
135     cell_length = velocity_verlet((double (*)[3]) position, (double (*)[3]) velocity, (double) lattice_param, (↩
            double) cell_length, (int) end_time, (double) dt, (int) n_cols, (int) nbr_atoms, \
136                     (bool) temp_scaling, (bool) press_scaling, (double) temp_eq, (double) press_eq, (bool) ↩
                            write_not_append, (double) tau_P, (double) tau_T);
137 }
138
139 void
140 H1_task4()
141 {
142     int nbr_atoms = 256; int n_rows = nbr_atoms; int n_cols = 3; int n_unitcells = 4;
143
144     // Initialising position and velocity arrrays
145     double position[nbr_atoms][n_cols];
146     double velocity[nbr_atoms][n_cols];
147     for(int ix = 0; ix < nbr_atoms; ix++){
148         for(int jx = 0; jx < n_cols; jx++){
149             velocity[ix][jx] = 0;
150         }
```

```
151     }
152
153     // Choosing lattice param
154     double lattice_param = 4.03; // True is around 4.0478    (Masahiko Morinaga, https://bit.ly/3ERRFt3)
155     double cell_length = 4 * lattice_param;
156
157     // Initialice and displace fcc
158     init_fcc((double (*)[3]) position, (int) n_unitcells, (double) lattice_param); // 4 unit cells in each ↩
            direction
159
160     displace_fcc((double (*)[3]) position, (int) n_unitcells, (double) lattice_param);
161
162     // Declaring parameters for velocity verlet.
163     // If temp/press_scaling = false scaling is turned off and scaling factors remains = 1
164     int end_time; double dt;
165     bool temp_scaling, press_scaling, write_not_append;
166     double temp_eq, press_eq, tau_T, tau_P;
167
168
169     // Melting run
170     end_time = 20; dt = 1e-3;
171     temp_scaling = true; press_scaling = true;
172     temp_eq = 5000; press_eq = 1; //773.15 K och 1 Bar
173     tau_T = 100*dt; tau_P = 300*dt; //50*dt;
174     write_not_append = true;
175
176     cell_length = velocity_verlet((double (*)[3]) position, (double (*)[3]) velocity, (double) lattice_param, (↩
            double) cell_length, (int) end_time, (double) dt, (int) n_cols, (int) nbr_atoms, \
177                     (bool) temp_scaling, (bool) press_scaling, (double) temp_eq, (double) press_eq, (bool) ↩
                            write_not_append, (double) tau_P, (double) tau_T);
178
179     // Cooling run
180     end_time = 20; dt = 1e-3;
181     temp_scaling = true; press_scaling = true;
182     temp_eq = 973.15; press_eq = 1; //773.15 K och 1 Bar
183     write_not_append = false;
184
185     cell_length = velocity_verlet((double (*)[3]) position, (double (*)[3]) velocity, (double) lattice_param, (↩
            double) cell_length, (int) end_time, (double) dt, (int) n_cols, (int) nbr_atoms, \
186                     (bool) temp_scaling, (bool) press_scaling, (double) temp_eq, (double) press_eq, (bool) ↩
                            write_not_append, (double) tau_P, (double) tau_T);
187
188
189     // Production run
190     end_time = 20; dt = 1e-3;
191     temp_scaling = false, press_scaling = false;
192     temp_eq = 973.15; press_eq = 1; //773.15 K och 1 Bar
193     write_not_append = true;
194
195     cell_length = velocity_verlet((double (*)[3]) position, (double (*)[3]) velocity, (double) lattice_param, (↩
            double) cell_length, (int) end_time, (double) dt, (int) n_cols, (int) nbr_atoms, \
196                     (bool) temp_scaling, (bool) press_scaling, (double) temp_eq, (double) press_eq, (bool) ↩
                            write_not_append, (double) tau_P, (double) tau_T);
197
198 }
199
200
201 void
202 H1_task6()
203 {
204     int nbr_atoms = 256; int n_rows = nbr_atoms; int n_cols = 3; int n_unitcells = 4;
205
206     // Initialising position and velocity arrrays
207     double position[nbr_atoms][n_cols];
208     double velocity[nbr_atoms][n_cols];
209     for(int ix = 0; ix < nbr_atoms; ix++){
210         for(int jx = 0; jx < n_cols; jx++){
211             velocity[ix][jx] = 0;
212         }
213     }
214
215     // Choosing lattice param
216     double lattice_param = 4.03; // True is around 4.0478    (Masahiko Morinaga, https://bit.ly/3ERRFt3)
217     double cell_length = 4 * lattice_param;
218
219     // Initialice and displace fcc
220     init_fcc((double (*)[3]) position, (int) n_unitcells, (double) lattice_param); // 4 unit cells in each ↩
            direction
221
222     displace_fcc((double (*)[3]) position, (int) n_unitcells, (double) lattice_param);
223
224     // Declaring parameters for velocity verlet.
```

```
    // If temp/press_scaling = false scaling is turned off and scaling factors remains = 1
    int end_time; double dt;
    bool temp_scaling, press_scaling, write_not_append;
    double temp_eq, press_eq, tau_T, tau_P;


    // Melting run
    end_time = 20; dt = 1e-3;
    temp_scaling = true; press_scaling = true;
    temp_eq = 5000; press_eq = 1; //773.15 K och 1 Bar
    tau_T = 100*dt; tau_P = 300*dt; //*dt; 50,5
    write_not_append = true;

    cell_length = velocity_verlet((double (*)[3]) position, (double (*)[3]) velocity, (double) lattice_param, (double) cell_length, (int) end_time, (double) dt, (int) n_cols, (int) nbr_atoms, \
                (bool) temp_scaling, (bool) press_scaling, (double) temp_eq, (double) press_eq, (bool) write_not_append, (double) tau_P, (double) tau_T);

    // Cooling run
    end_time = 20; dt = 1e-3;
    temp_scaling = true; press_scaling = true;
    temp_eq = 973.15; press_eq = 1; //773.15 K och 1 Bar
    write_not_append = false;

    cell_length = velocity_verlet((double (*)[3]) position, (double (*)[3]) velocity, (double) lattice_param, (double) cell_length, (int) end_time, (double) dt, (int) n_cols, (int) nbr_atoms, \
                (bool) temp_scaling, (bool) press_scaling, (double) temp_eq, (double) press_eq, (bool) write_not_append, (double) tau_P, (double) tau_T);


    // Production run
    end_time = 20; dt = 1e-3;
    temp_scaling = false, press_scaling = false;
    temp_eq = 973.15; press_eq = 1; //773.15 K och 1 Bar
    write_not_append = true;

    int number_of_bins = 150;
    double *radial_distribution_vector = calloc(sizeof(double),  number_of_bins);
    char filename_radial_dist[] = {"../csv/radial_distribution.csv"};
    double normalisation_factor_radial_dist = (double) nbr_atoms*end_time/(dt);


    cell_length = velocity_verlet_with_radial((double (*)[3]) position, (double (*)[3]) velocity, (double) lattice_param, (double) cell_length, (int) end_time, (double) dt, (int) n_cols, (int) nbr_atoms, \
                (bool) temp_scaling, (bool) press_scaling, (double) temp_eq, (double) press_eq, (bool) write_not_append, (double) tau_P, (double) tau_T, radial_distribution_vector, number_of_bins);


    for(int bin = 0; bin<number_of_bins; ++bin)
    {

        //also need to divide by number of time steps
        radial_distribution_vector[bin] /=normalisation_factor_radial_dist;
    }

    bool is_empty = true;
    save_vector_to_csv(radial_distribution_vector, number_of_bins, filename_radial_dist, is_empty);
    free(radial_distribution_vector);
}
```

A.2 Calculating equilibrium lattice parameter: `try_lattice_constants.c`

```
double
try_lattice_constants(int N_atoms, int n_rows, int n_cols){
    // Initializing values needed for task
    double E_pot = 0; int n_lattice_params = 8;
    char filename_result[] = {"try_lattice_constants.csv"};
    double lattice_params[n_lattice_params]; double lattice_param_init = 4.05; // 4.0478; // Lattice_param, denoted a0 in document. Should be 4.0478    (Masahiko Morinaga, https://bit.ly/3ERRFt3)

    // Set print_q=true if we want to print the lattice_parameters code is looping through.
    // Lattice params is an array around lattice_param_init with lattice_param_spacing
    bool print_q = false; double lattice_param_spacing = 0.05;
    if(print_q = true){
        printf("n/2-ix, lattice_param \n");
        for(int ix = 0; ix < n_lattice_params; ix++){
            printf("%i, ", n_lattice_params/2-ix);
```

```
            lattice_params[ix] = lattice_param_init - (n_lattice_params/2-ix)*lattice_param_spacing;
            printf("%f\n", lattice_params[ix]);
        }
        printf("\n");
    } else{
        for(int ix = 0; ix < n_lattice_params; ix++){
            lattice_params[ix] = lattice_param_init - (n_lattice_params/2-ix)*lattice_param_spacing;
        }
        printf("\n");
    }

    double smallest_E_potperunitcell = 0; double smallest_lattice_param = 0;
    // For loop to run over lattice parameters
    for(int ix = 0; ix < n_lattice_params; ix++){
        // Initialize matric and update values
        double pos_matrix[256][3];//double pos_matrix[4*256][3];
        double lattice_param = lattice_params[ix];
        double lattice_volume = pow(lattice_param, 3);

        // Retrieve fcc with already made function
        init_fcc((double (*)[3]) pos_matrix, (int) 4, (double) lattice_param); // 4 unit cells in each direction

        // Retrieve potential energy with ready made function
        double cell_length = 4 * lattice_param; // 4-unit cells
        E_pot = get_energy_AL((double (*)[3]) pos_matrix, (double) cell_length, (int) N_atoms);

        // Scaling result with number of unit cells and initializing result vector
        double E_pot_per_unitcell = E_pot/pow(4,3);
        double result_vec[] = {ix, lattice_param, lattice_volume, E_pot, E_pot_per_unitcell};

        //Saving smallest lattice param if it results in lowest potential energy per unit cell
        if(E_pot_per_unitcell<smallest_E_potperunitcell){
            smallest_E_potperunitcell = E_pot_per_unitcell;
            smallest_lattice_param = lattice_param;
        }

        // Saving results to csv file "try_lattice_constants.csv"
        if(ix==0){
            save_vector_to_csv(result_vec, 5, filename_result, true); // true -> fopen with "w"
        } else {
            save_vector_to_csv(result_vec, 5, filename_result, false); // false -> fopen with "a"
        }

        //Printing results in terminal if print_book is set to true
        bool print_bool = true;
        if(print_bool == true) {
            printf("Lattice_volume: %f\n", lattice_volume);
            printf("E_pot: %f\n", E_pot);
            printf("E_pot_perunitcell: %f\n", E_pot_per_unitcell);
            printf("\n");
        }
    }

    return smallest_lattice_param;
}
```

A.3 Velocity-Verlet-algorithm, with get_radial_distribution: `velocity`$_{v}$*erlet.c*

```

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

#include "lattice.h"
#include "potential.h"
#include "tools.h"
#include "try_lattice_constants.h"

#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
#include <time.h>

double
velocity_verlet(double positions[][3], double v[][3], double lattice_param, double cell_length, int end_time, \
double dt, int n_cols, int nbr_atoms, bool temp_scaling, bool press_scaling, double temp_eq, double press_eq, ↩
     bool write_not_append, \
double tau_P, double tau_T)
```

```
{
    // Initialize variables
    double cell_volume = pow(cell_length, 3.);
    double lattice_volume = pow(lattice_param, 3.);
    double aluminium_amu = 26.98; double m_asu = 9649;
    double aluminium_asu = aluminium_amu/m_asu; //Mass in atomic simulation units

    double E_kinetic = 0; double E_kinetic_per_unitcell = 0;
    double E_potential = 0; double E_potential_per_unitcell = 0;
    double E_total_per_unitcell = 0;

    double virial = 0; double virial_per_unitcell = 0;
    double temp_inst = 0; double temp_inst_per_unitcell = 0;
    double press_inst = 0; double press_inst_per_unitcell = 0;
    double kB = 8.61733 * 1e-5; // Boltzmann constant in eV/K

    //char *filename_result, *filename_pos, *filename_param;
    char filename_result[] = {"../csv/vel_verlet_eq.csv"};
    char filename_pos[] = {"../csv/position_track_eq.csv"};
    char filename_param[] = {"../csv/parameters_eq.csv"};

    char filename_result_prod[] = {"../csv/vel_verlet_prod.csv"};
    char filename_pos_prod[] = {"../csv/position_track_prod.csv"};
    char filename_param_prod[] = {"../csv/parameters_prod.csv"};
    char filename_radial_distribution[] = {"../csv/radial_distribution.csv"};
    bool is_write;

    //Creating empty arrays
    //double v[nbr_atoms][n_cols];
    double f[nbr_atoms][n_cols];

    for(int ix = 0; ix < nbr_atoms; ix++){
        for(int jx = 0; jx < n_cols; jx++){
            //v[ix][jx] = 0;
            f[ix][jx] = 0;
        }
    }

    // Variables for duration of measurement
    int n_timesteps = end_time / dt;

    // Declaring scaling variables, if temp/press_scaling = false scaling is turned off and alpha_T/P just ↩
        remains 1
    double alpha_T = 1; double alpha_P = 1;

    //Velocity verlet algorithm as in E1
    get_forces_AL((double (*)[3]) f, (double (*)[3]) positions, (double) cell_length, (int) nbr_atoms);


    for(int tx = 1; tx < n_timesteps + 1; tx++)
    {
        /* v(t+dt/2) */
        for(int ix = 0; ix < nbr_atoms; ix++)
        {
            for(int jx = 0; jx < n_cols; jx++)
            {
                v[ix][jx] += 0.5 * dt * f[ix][jx] / aluminium_asu;
                //printf("v[%i][%i] %f\n",ix, jx, v[ix][jx]);
            }
        }

        /* q(t+dt) */
        for(int ix = 0; ix < nbr_atoms; ix++)
        {
            for(int jx = 0; jx < n_cols; jx++)
            {
                // alpha_P for scaling. If press_scaling==false this is 1.
                positions[ix][jx] += dt * v[ix][jx];
                positions[ix][jx] *= cbrt(alpha_P);
                //positions[ix][jx] *= pow(alpha_P, (double) 1./3.);
            }
        }

        // After scaling positions lattice_parameter is scaled to change pressure
        cell_length *= cbrt(alpha_P); cell_volume = cell_length * cell_length * cell_length;


        /* a(t+dt) */
        get_forces_AL((double (*)[3]) f, (double (*)[3]) positions, (double) cell_length, (int) nbr_atoms);

        // Resetting kinetic energy for every timestep to ensure summing over particles and not over timesteps
        E_kinetic = 0;
```

```
        /* v(t+dt) */
        for(int ix = 0; ix < nbr_atoms; ix++)
        {
            for(int jx = 0; jx < n_cols; jx++)
            {
                // alpha_T for scaling. If temp_scaling==false this is 1.
                v[ix][jx] += 0.5 * dt * f[ix][jx] / aluminium_asu;
                v[ix][jx] *= sqrt(alpha_T);
                E_kinetic += 0.5 * aluminium_asu * v[ix][jx] * v[ix][jx];
            }
        }

        // Calculate potential, kinetic and total energy per unitcell
        E_potential = get_energy_AL((double (*)[3]) positions, (double) cell_length, (int) nbr_atoms);
        E_potential_per_unitcell = E_potential / pow(4.,3.);
        E_kinetic_per_unitcell = E_kinetic / pow(4.,3.);
        E_total_per_unitcell = E_potential_per_unitcell + E_kinetic_per_unitcell;

        // Calculate virial to use for pressure calculation
        virial = get_virial_AL((double (*)[3]) positions, (double) cell_length, (int) nbr_atoms);
        virial_per_unitcell = virial / pow(4.,3.);

        double N = 256;
        double N1 = 4;
        temp_inst_per_unitcell = 2 / (3 * N * kB) * E_kinetic;
        //temp_inst_per_unitcell = 2 / (3 * N1 * kB) * E_kinetic_per_unitcell;

        // Change scaling parameter for temperature
        if(temp_scaling == true){

            alpha_T = 1 + 2 * dt / tau_T * (temp_eq - temp_inst_per_unitcell)/temp_inst_per_unitcell;


         // Calculate pressure in eV/  ^3
        press_inst_per_unitcell = ((N * kB * temp_inst_per_unitcell) + virial) / cell_volume; //lattice_volume;

        //press_inst_per_unitcell = ((N * kB * temp_inst_per_unitcell) + virial_per_unitcell) / cell_volume; //lattice_volume;

        press_inst_per_unitcell *= 1.60219*1e2*1e4; // 1eV/  ^3 = 160.2 Gpa, 1 GPa = 10^4 Bar

        // Changing scaling parameter for pressure
        if(press_scaling == true){

            double kappa_T = 0.01385*1e-4; // Neg eller pos?

            // Unsure if plus or minus. Scaling with "correct" sign seams to change pressure in wrong direction
            alpha_P = 1 - kappa_T * dt / tau_P * (press_eq - press_inst_per_unitcell);
        }

        // Creating vectors so to save results in csv files. Can be plotted with python files plot_energy.py and plot_position_track.py
        double parameter_vec[] = {end_time, dt, lattice_param, temp_scaling, press_scaling, temp_eq, press_eq, tau_T, tau_P};
        double result_vec[] = {tx*dt, cell_length, E_potential_per_unitcell, E_kinetic_per_unitcell, E_total_per_unitcell, temp_inst_per_unitcell, press_inst_per_unitcell, alpha_T, alpha_P};
        double position_track_vec[] = {tx*dt,positions[23][0], positions[23][1], positions[23][2],\
                                       positions[135][0], positions[135][1], positions[135][2],\
                                       positions[189][0], positions[189][1], positions[189][2],\
                                       temp_inst_per_unitcell};

        // Saving results to csv files
        if(tx == 1){is_write = write_not_append;} else {is_write = false;};
        if(temp_scaling == true || press_scaling == true)
        {   save_vector_to_csv(result_vec, 9, filename_result, is_write); // true -> fopen with "w"
            save_vector_to_csv(position_track_vec, 10, filename_pos, is_write); // false -> fopen with "a"
            if(tx == n_timesteps){
                save_vector_to_csv(parameter_vec, 9, filename_param, is_write);
            }
            printf("Calibration: Inst. Temp, Press at t = [%i]: %f,   %f\n", tx, temp_inst_per_unitcell, press_inst_per_unitcell);
        } else {
            save_vector_to_csv(result_vec, 9, filename_result_prod, is_write); // true -> fopen with "w"
            save_vector_to_csv(position_track_vec, 10, filename_pos_prod, is_write); // false -> fopen with "a"

            if(tx == n_timesteps){
                save_vector_to_csv(parameter_vec, 9, filename_param_prod, is_write);
            }
            printf("Production: Inst. Temp, Press at t = [%i]: %f,   %f\n", tx, temp_inst_per_unitcell, press_inst_per_unitcell);
        }
```

```
        // Printing temperature for each timestep to keep track during longer measurements
        //printf("Inst. Temp, Press at t = [%i]: %f,   %f\n", tx, temp_inst_per_unitcell, press_inst_per_unitcell);
    }
    return cell_length;
}


//

double
velocity_verlet_with_radial(double positions[][3], double v[][3], double lattice_param, double cell_length, int end_time, \
double dt, int n_cols, int nbr_atoms, bool temp_scaling, bool press_scaling, double temp_eq, double press_eq, bool write_not_append, \
double tau_P, double tau_T, double *radial_histogram_vector, int number_of_bins)
{
    // Initialize variables
    double cell_volume = pow(cell_length, 3);
    double lattice_volume = pow(lattice_param, 3);
    double aluminium_amu = 26.98; double m_asu = 9649;
    double aluminium_asu = aluminium_amu/m_asu; //Mass in atomic simulation units

    double E_kinetic = 0; double E_kinetic_per_unitcell = 0;
    double E_potential = 0; double E_potential_per_unitcell = 0;
    double E_total_per_unitcell = 0;

    double virial = 0; double virial_per_unitcell = 0;
    double temp_inst = 0; double temp_inst_per_unitcell = 0;
    double press_inst = 0; double press_inst_per_unitcell = 0;
    double kB = 8.61733 * 1e-5; // Boltzmann constant in eV/K

    //char *filename_result, *filename_pos, *filename_param;
    char filename_result[] = {"../csv/vel_verlet_eq.csv"};
    char filename_pos[] = {"../csv/position_track_eq.csv"};
    char filename_param[] = {"../csv/parameters_eq.csv"};

    char filename_result_prod[] = {"../csv/vel_verlet_prod.csv"};
    char filename_pos_prod[] = {"../csv/position_track_prod.csv"};
    char filename_param_prod[] = {"../csv/parameters_prod.csv"};
    char filename_radial_distribution[] = {"../csv/radial_distribution.csv"};
    bool is_write;

    //Creating empty arrays
    //double v[nbr_atoms][n_cols];
    double f[nbr_atoms][n_cols];

    for(int ix = 0; ix < nbr_atoms; ix++){
        for(int jx = 0; jx < n_cols; jx++){
            //v[ix][jx] = 0;
            f[ix][jx] = 0;
        }
    }

    // Variables for duration of measurement
    int n_timesteps = end_time / dt;

    // Declaring scaling variables, if temp/press_scaling = false scaling is turned off and alpha_T/P just remains 1
    double alpha_T = 1; double alpha_P = 1;

    //Velocity verlet algorithm as in E1
    get_forces_AL((double (*)[3]) f, (double (*)[3]) positions, (double) cell_length, (int) nbr_atoms);


    get_radial_dist_AL(number_of_bins, radial_histogram_vector, (double (*)[3])positions, cell_length, nbr_atoms);

    for(int tx = 1; tx < n_timesteps + 1; tx++)
    {
        /* v(t+dt/2) */
        for(int ix = 0; ix < nbr_atoms; ix++)
        {
            for(int jx = 0; jx < n_cols; jx++)
            {
                v[ix][jx] += 0.5 * dt * f[ix][jx] / aluminium_asu;
                //printf("v[%i][%i] %f\n",ix, jx, v[ix][jx]);
            }
        }

        /* q(t+dt) */
        for(int ix = 0; ix < nbr_atoms; ix++)
```

```
        {
            for(int jx = 0; jx < n_cols; jx++)
            {
                // alpha_P for scaling. If press_scaling==false this is 1.
                positions[ix][jx] += dt * v[ix][jx];
                positions[ix][jx] *= cbrt(alpha_P);
                //positions[ix][jx] *= pow(alpha_P, (double) 1/3);
            }
        }

        // After scaling positions lattice_parameter is scaled to change pressure
        cell_length *= cbrt(alpha_P); cell_volume = cell_length*cell_length*cell_length;


        /* a(t+dt) */
        get_forces_AL((double (*)[3]) f, (double (*)[3]) positions, (double) cell_length, (int) nbr_atoms);
        get_radial_dist_AL(number_of_bins, radial_histogram_vector, (double (*)[3])positions, cell_length, ↩
             nbr_atoms);

        // Resetting kinetic energy for every timestep to ensure summing over particles and not over timesteps
        E_kinetic = 0;

        /* v(t+dt) */
        for(int ix = 0; ix < nbr_atoms; ix++)
        {
            for(int jx = 0; jx < n_cols; jx++)
            {
                // alpha_T for scaling. If temp_scaling==false this is 1.
                v[ix][jx] += 0.5 * dt * f[ix][jx] / aluminium_asu;
                v[ix][jx] *= sqrt(alpha_T);
                E_kinetic += 0.5 * aluminium_asu * v[ix][jx] * v[ix][jx];
            }
        }

        // Calculate potential, kinetic and total energy per unitcell
        E_potential = get_energy_AL((double (*)[3]) positions, (double) cell_length, (int) nbr_atoms);
        E_potential_per_unitcell = E_potential / 64;// pow(4.,3.);
        E_kinetic_per_unitcell = E_kinetic / 64; //pow(4.,3.);
        E_total_per_unitcell = E_potential_per_unitcell + E_kinetic_per_unitcell;

        // Calculate virial to use for pressure calculation
        virial = get_virial_AL((double (*)[3]) positions, (double) cell_length, (int) nbr_atoms);
        virial_per_unitcell = virial / 64;//pow(4.,3.);

        double N = 256;

        // Calculate temperature (4 atoms in unit cell, kB in eV/K)
        temp_inst_per_unitcell = 2 / (3 * N * kB) * E_kinetic;
        //temp_inst_per_unitcell = 2 / (3 * N1 * kB) * E_kinetic_per_unitcell;

        // Change scaling parameter for temperature
        if(temp_scaling == true){

            alpha_T = 1 + 2 * dt / tau_T * (temp_eq - temp_inst_per_unitcell)/temp_inst_per_unitcell;
        }
        press_inst_per_unitcell = ((N * kB * temp_inst_per_unitcell) + virial) / cell_volume; //lattice_volume;
        press_inst_per_unitcell *= 1.60219*1e2*1e4; // 1eV/  ^3 = 160.2 gPA

        // Changing scaling parameter for pressure
        if(press_scaling == true){

            // Isothermal compressability for aluminium in Gpa^-1
            double kappa_T = 0.01385*1e-4;

            alpha_P = 1 - kappa_T * dt / tau_P * (press_eq - press_inst_per_unitcell);
        }

        // Creating vectors so to save results in csv files. Can be plotted with python files plot_energy.py and ↩
             plot_position_track.py
        double parameter_vec[] = {end_time, dt, lattice_param, temp_scaling, press_scaling, temp_eq, press_eq, ↩
             tau_T, tau_P};
        double result_vec[] = {tx*dt, cell_length, E_potential_per_unitcell, E_kinetic_per_unitcell, ↩
             E_total_per_unitcell, temp_inst_per_unitcell, press_inst_per_unitcell, alpha_T, alpha_P};
        double position_track_vec[] = {tx*dt,positions[23][0], positions[23][1], positions[23][2],\
                                       positions[135][0], positions[135][1], positions[135][2],\
                                       positions[189][0], positions[189][1], positions[189][2],\
                                       temp_inst_per_unitcell};

        // Saving results to csv files
        if(tx == 1){is_write = write_not_append;} else {is_write = false;};
        if(temp_scaling == true || press_scaling == true)
        {   save_vector_to_csv(result_vec, 9, filename_result, is_write); // true -> fopen with "w"
```

```
            save_vector_to_csv(position_track_vec, 10, filename_pos, is_write); // false -> fopen with "a"
            if(tx == n_timesteps){
                save_vector_to_csv(parameter_vec, 9, filename_param, is_write);
            }
            printf("Calibration: Inst. Temp, Press at t = [%i]: %f,   %f\n", tx, temp_inst_per_unitcell, ↩
                 press_inst_per_unitcell);
        } else {
            save_vector_to_csv(result_vec, 9, filename_result_prod, is_write); // true -> fopen with "w"
            save_vector_to_csv(position_track_vec, 10, filename_pos_prod, is_write); // false -> fopen with "a"

            if(tx == n_timesteps){
                save_vector_to_csv(parameter_vec, 9, filename_param_prod, is_write);
            }
            printf("Production: Inst. Temp, Press at t = [%i]: %f,   %f\n", tx, temp_inst_per_unitcell, ↩
                 press_inst_per_unitcell);
        }
       }
    return cell_length;
}

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

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

  rcut = 6.06; // Embedded atom method potential.
  rcut_sq = rcut * rcut;


  cell_length_inv = 1 / cell_length;
  cell_length_sq = cell_length * cell_length;
  bin_length= cell_length/(double)number_of_bins;

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

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

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

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

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

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

      /*Add position into bin depending on radial size*/
      rij = sqrt( rij_sq );

      int bin = (int) floor(  rij/bin_length + 0.5);

      radial_histogram_vector[bin] +=1;

    }
```

```
  }

  free(sx); free(sy); free(sz); sx = NULL; sy = NULL; sz = NULL;
}
```

B Python functions for plotting

B.1 Main plotting function: `plot_everything.py`

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

# set default figure size
plt.rcParams["figure.figsize"] = [8, 6]

SMALL_SIZE = 15
MEDIUM_SIZE = 18
BIGGER_SIZE = 18

plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=MEDIUM_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)

for idx, val in enumerate(["eq", "prod"]):
    print(idx)
    print(val)
    str = val

    # load data from file
    array = np.genfromtxt(f'../csv/vel_verlet_{str}.csv', delimiter=',', skip_header=1)
    parameters = np.genfromtxt(f'../csv/parameters_{str}.csv', delimiter=',')
    pos_array = np.genfromtxt(f'../csv/position_track_{str}.csv', delimiter=',', skip_header=1)

    end_time = parameters[-1,0]
    dt = parameters[-1,1]
    lattice_param = parameters[-1,2]
    temp_scaling = parameters[-1,3]
    press_scaling = parameters[-1,4]
    temp_eq = parameters[-1,5]
    press_eq = parameters[-1,6]
    tau_T = parameters[-1,7]
    tau_P = parameters[-1,8]

    t = dt * np.linspace(0,len(array[:,1]), len(array[:,1]))
    cell_length = array[:,1]
    lattice_length = cell_length/4
    e_pot = array[:,2]
    e_kin = array[:,3]
    e_tot = array[:,4]
    temp = array[:,5]
    press = array[:,6]

    q1x = pos_array[:,1]
    q1y = pos_array[:,2]
    q1z = pos_array[:,3]
    q2x = pos_array[:,4]
    q2y = pos_array[:,5]
    q2z = pos_array[:,6]
    q3x = pos_array[:,7]
    q3y = pos_array[:,8]
    q3z = pos_array[:,9]

    # create figure and axes for energy plot
    fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(14, 5))

    # plot potential energy on first subplot
    axes[0].plot(t, e_pot, label='Potential energy', color="b")
    axes[0].set_title(f'Potential Energy, dt={dt} [ps]')

    # plot kinetic energy on second subplot
    axes[1].plot(t, e_kin, label='Kinetic energy', color="r")
```

```
    axes[1].set_title(f'Kinetic Energy, dt={dt} [ps]')

    # plot total energy on third subplot
    axes[2].plot(t, e_tot, label='Total energy', color="g")
    axes[2].set_title(f'Total Energy, dt={dt} [ps]')
    axes[2].ticklabel_format(useOffset=False)

    for idx in range(3):
        axes[idx].set_xlabel('Time [ps]')
        axes[idx].set_ylabel('Energy [eV/unit cell]')

    plt.tight_layout()
    plt.savefig(f'plots/energy_{str}.png')

    figlattice , ax_lattice = plt.subplots(1,1)

    ############# LATTICE ################
    # plot energy data
    ax_lattice.plot(t, lattice_length, label='Lattice parameter')

    average_lattice = np.mean(lattice_length)

    # add dashed line for the average energy
    ax_lattice.annotate(f'a$_{{0}}$ = {lattice_length[-1]:.3f}', xy=(t[-1], lattice_length[-1]), xytext=(-50, ↩
         -50),
                textcoords='offset pixels', arrowprops=dict(arrowstyle='->', color='k'))

    # set labels and title
    ax_lattice.set_xlabel('Time [ps]')
    ax_lattice.set_ylabel('Lattice parameter [  ]')
    ax_lattice.set_title(f'Lattice parameter, dt={dt}')

    plt.legend()
    plt.tight_layout()
    plt.savefig(f'plots/lattice_{str}.png')



    ####### POSITIONS ###################
    fig_pos, ax_pos = plt.subplots(1, 3, figsize=(14, 5))

    # Iterate over dimensions
    for i, dim in enumerate(["X", "Y", "Z"]):
        # Plot each dimension in a subplot
        ax_pos[i].plot(t, eval(f"q1{dim.lower()}"), label="q1")
        ax_pos[i].plot(t, eval(f"q2{dim.lower()}"), label="q2")
        ax_pos[i].plot(t, eval(f"q3{dim.lower()}"), label="q3")

        # Set x- and y-labels and title
        ax_pos[i].set_xlabel("Time [ps]")
        ax_pos[i].set_ylabel(f"{dim}-coordinate [  ]")
        ax_pos[i].set_title(f"{dim}-coordinate, dt={dt}")

        # Add legend
        ax_pos[i].legend(loc='lower right')

    # Adjust layout and save figure
    plt.tight_layout()
    plt.savefig(f"plots/position_track_{str}.png")


    ########### PRESSURE #############
    figP , axP = plt.subplots(1,1)
    axP.plot(t, press, label='Pressure')

    average_pressure = np.mean(press)

    # add dashed line for the average energy
    if(str == "prod"):
        print("Hej")
        axP.axhline(
            average_pressure,
            linestyle='--',
            color='r',
            linewidth=4,
            label=f'P$_{{average}}$ = {average_pressure:.2f} [Bar]'
        )

        #axP.annotate(f'P$_{{average}}$ = {average_pressure:.2f}', xy=(t[-1], average_pressure), xytext=(-150, ↩
             150),
        #            textcoords='offset pixels', arrowprops=dict(arrowstyle='->', color='k'))
```

```
    axP.set_xlabel('Time [ps]')
    axP.set_ylabel('Pressure [Bar]')
    axP.set_title(f'Pressure, dt={dt}')

    plt.legend()
    plt.tight_layout()
    plt.savefig(f'plots/pressure_{str}.png')
    #plt.show()



    ########### TEMPTERATURE #############

    average_temperature = np.mean(array[:,5])

    fig2 , axT = plt.subplots(1,1)
    axT.plot(t, temp, label='Temperature')

    # add dashed line for the average energy
    if(str == "prod"):
        axT.axhline(
            average_temperature,
            linestyle='--',
            color='r',
            linewidth=4,
            label=f'T$_{{average}}$ = {average_temperature:.2f} [K]'
        )

        #axT.annotate(f'T$_{{average}}$ = {average_temperature:.2f}', xy=(t[-1], average_temperature), xytext=(-110, 200),
        #            textcoords='offset pixels', arrowprops=dict(arrowstyle='->', color='k'))

    plt.legend()
    axT.set_xlabel('Time [ps]')
    axT.set_ylabel('Temperature [K]')
    axT.set_title(f'Temperature, dt={dt}')

    plt.legend()
    plt.tight_layout()
    plt.savefig(f'plots/temperature_{str}.png')
```

B.2 Plotting function for task 6 with calculation of heat capacity and coordination number: `plot_task6.py`

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy as sp
sns.set()

# set default figure size
plt.rcParams["figure.figsize"] = [8, 6]

SMALL_SIZE = 13
MEDIUM_SIZE = 15
BIGGER_SIZE = 15

plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=MEDIUM_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE)    # legend fontsize
plt.rc('figure', titlesize=BIGGER_SIZE)


Cell_length = 4*4.03; number_particles = 256
array = np.genfromtxt(f'../csv/radial_distribution.csv', delimiter=',', skip_header=0)
number_of_bins = len(array)
def N_ideal(bin_arr, Cell_length, Number_of_particles):
    Number_of_bins = len(bin_arr)

    delta_r = Cell_length/Number_of_bins
    V= Cell_length**3

    N_ideal = (Number_of_particles-1)*4*np.pi/(3*Cell_length**3) * (np.square(bin_arr) -3*bin_arr+1)*delta_r**3

    return N_ideal
```

```

bin_array = np.arange(number_of_bins)
ideal = N_ideal(bin_array, Cell_length, number_particles)
delta_r = Cell_length/number_particles
fig, ax_radial = plt.subplots(1,1)
array = (array/ideal)
max =np.argmax(array)
upper = 100
ax_radial.plot(bin_array*delta_r, array, label='$g(r)$')
ax_radial.set_xlim(0,8)

minima =np.argmin(array[max:upper])
max = np.argmax(array)

index_r_m = max+minima

plt.rcParams['xtick.labelsize'] = 30
radiusvec =bin_array*delta_r
ax_radial.scatter(radiusvec[index_r_m], array[index_r_m], color='r', label="$r_m$", s=40)
ax_radial.set_title(r"Radial distribution function $g(r)$", fontsize=15)
ax_radial.set_xlabel("$r$ [  ]", fontsize=15)
ax_radial.set_ylabel(r"$g(r)$", fontsize=15)
ax_radial.legend(fontsize=15)
plt.tight_layout()

integrand =[]

for i,r in enumerate(bin_array[:index_r_m]*delta_r):
    print(r,i, array[i])
    integrand.append(r**2* 4*np.pi* array[i]*(256/Cell_length**3))

coordination_number = sp.integrate.trapezoid(bin_array[:index_r_m]*delta_r, integrand)
print(coordination_number)
fig.savefig(f'plots/radial.png')
```

C Provided functions

C.1 Potential, included in course material: `potentials.c`

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

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


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

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


/* Evaluates the spline in x. */

double splineEval(double x, const double *table,int m) {
    /* int m = mxGetM(spline), i, k;*/
```

```
    int i, k;

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

    int k_lo = 0, k_hi = m;

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

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

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

    return result;
}

/* Evaluates the derivative of the spline in x. */

double splineEvalDiff(double x, const double *table, int m) {
    /*int m = mxGetM(spline), i, k;
   double *table = mxGetPr(spline);
   */
  int i, k;
  double result;

    int k_lo = 0, k_hi = m;

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

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

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

    return result;
}

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

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

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

```

```
  rcut = 6.06;
  rcut_sq = rcut * rcut;

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

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

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

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

    densityi = density[i];

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

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

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

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

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

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

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

    densityi = density[i];

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

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

```

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

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

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

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

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

}

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

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

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

  rcut = 6.06;
  rcut_sq = rcut * rcut;

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

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

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

  energy = 0;

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

    densityi = density[i];

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

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

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

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

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

      }
    }
    density[i] = densityi;
  }

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

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

  return(energy);

}

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

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

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

  rcut = 6.06;
  rcut_sq = rcut * rcut;

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

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

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

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

    densityi = density[i];

        /* Loop over other atoms. */
        for (j = i + 1; j < nbr_atoms; j++) {
      /* Periodically translate atom j to positive quadrants and calculate distance to it. */
```

```
            sxij = sxi - (sx[j] - floor(sx[j]));
            syij = syi - (sy[j] - floor(sy[j]));
            szij = szi - (sz[j] - floor(sz[j]));

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

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

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

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

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

  virial = 0;

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

    densityi = density[i];

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

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

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

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

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

        virial += force * rij_sq;
            }
    }
  }

  virial /= 3.0;

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

  return(virial);

}
```

C.2 Lattice, included in course material: `lattice.c`

```c
/*
H1lattice.c
Program that arranges atoms on a fcc lattice.
Created by Anders Lindman on 2013-03-15.
*/

#include <stdio.h>

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

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