H1b: MD Simulation - Dynamic Properties

Tomas Lundberg and Jonas H. Fritz

December 8, 2021

Task №	Points	Avail. points
$\sum$		

Introduction

In this report we numerically investigate a 4x4x4 supercell of fcc aluminum with periodic boundary conditions. With the use of the velocity Verlet algorithm we solve the equations of motion for time periods on the order of 10 ps. Furthermore, we employ an equilibration technique and investigate aluminum in both the solid and liquid phase. For both phases we proceed to calculate the mean square displacement and velocity correlation functions; the latter using two different approaches.

Problem 1

To get familiar with the setup, we start by initializing a 4x4x4 supercell of fcc cells, that contain 4 atoms each, i.e. 256 atoms. The routine to set up the lattice is provided, and takes the number of unit cells in each parameter and the lattice constant a_0 as parameters. We vary the lattice constant and calculate the potential energy for each lattice using the provided function. The potential energy as function of lattice constant is shown in fig. 1. We fit a quadratic function to obtain its minimum, which corresponds to the equilibrium lattice constant at temperature $T = 0\,\mathrm{K}$, a_0^*. We obtain $a_0^* = 4.0318\,\mathrm{\AA}$, which corresponds to a unit cell volume of $V = 65.54\,\mathrm{\AA}^3$. This seem reasonable since the lattice constant at room temperature is 4.046 Å [2].

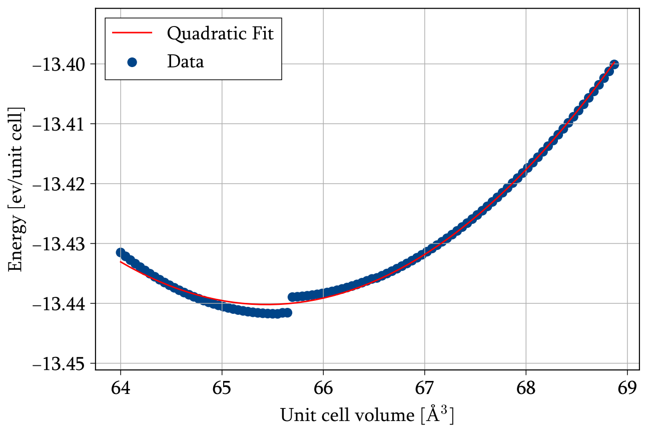

Figure 1: Potential energy of the initialized lattice for different lattice constants. We use a quadratic fit to obtain the energy minimum at a unit cell size of $V = 65.54\,\mathrm{\AA}^3$.

Problem 2

We now start considering the dynamics of the system by adding energy and implementing the velocity Verlet algorithm to integrate the equations of motion [3]. As a first step we randomly perturb each coordinate of each particle by 6.5% of the lattice parameter. Then we simulate the dynamics of the system for 10 ps with 20 different time step size $\Delta t_i = 0.001 \cdot 1.25^i$ ps for $i = 0, 1, 2, \ldots, 19$ in order to evaluate a suitable Δt. We calculate total energy over to 10 ps and compute both the standard deviation and the absolute mean for each trajectory, to see at which point energy conservation breaks down. This is show in fig. 2. We see that the mean of total energy drastically changes above $\Delta t = 0.01$ ps, but the standard deviation already slowly increases above $\Delta t = 0.001$ ps. With this in mind we settle for a $\Delta t = 0.001$ ps for the following simulations.

We now run the simulation for 5 ps, i.e 5000 iterations, with this Δt and also record kinetic, potential and total energy and calculate the instantaneous temperature. The results of the simulation are shown in fig. 3. We can see that with our initial conditions, the system equilibrates at around $T = 770\,\mathrm{K}$. Since we chose Δt such that the total energy should be conserved it is no surprise that this is also the case in fig. 3. Furthermore, because of the perturbation to the initial positions in the lattice, the system has been given some initial energy, i.e. a temperature. These initial perturbations lead to the constant exchange of kinetic and potential energy between the atoms in the lattice. Thus, the fluctuations we see in the kinetic and potential energy have a physical interpretation, and the only relevent measure for choosing the time step is total energy conservation. If we would have had larger initial perturbations, the temperature would have been higher and equivalently the kinetic energy would have been higher. The potential energy would have fluctuated more, although around a similar value as now. The total energy will, of course, always be conserved, if the time step is chosen appropriately.

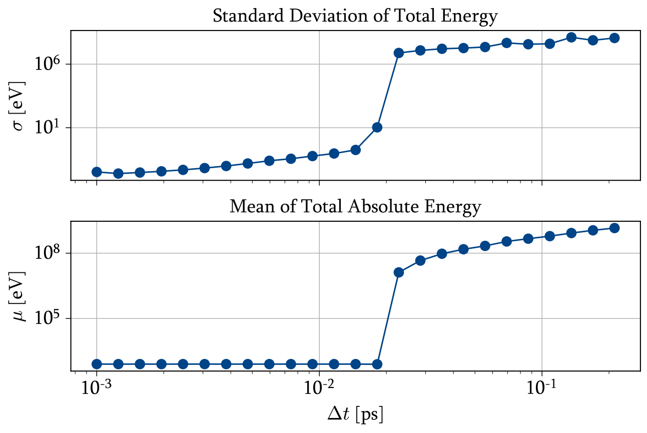

Figure 2: Standard deviation and mean total energy for different choices of simulation time step Δt. The standard deviation increases continuously, so we settle for $\Delta t = 0.001$.

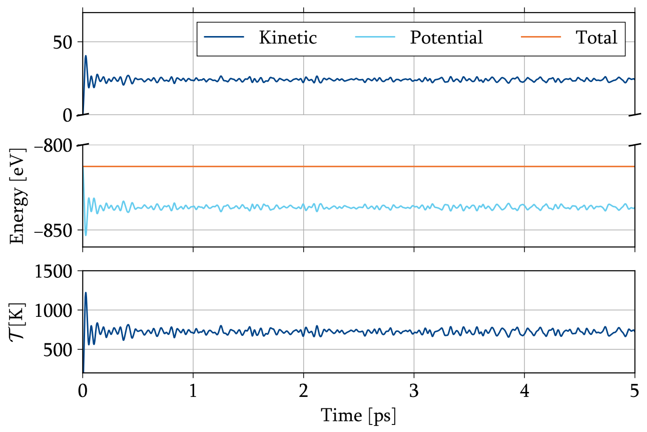

Figure 3: Kinetic (dark blue), potential (light blue) and total energy (orange) energy along the trajectory. Kinetic and potential energy fluctuate,, but since the velocity verlet algorithm preserves the symplectic structure of the equations of motion, total energy is conserved. Below is the instantaneous temperature along the trajectory, which follows the fluctuations of kinetic energy.

Problem 3

In this problem we implement the rescaling of positions and velocities to speed up the equilibration process [3]. We want equilibration to take place exponentially in time, which leads to

$$\begin{aligned} v_i^{\text{new}} &= \alpha_T^{1/2} v_i^{\text{old}}\,, \\ r_i^{\text{new}} &= \alpha_P^{1/2} r_i^{\text{old}}\,, \end{aligned} \tag{1}$$

with

$$\begin{aligned}\alpha_T(t) &= 1 + \frac{2\Delta t}{\tau_T}\frac{T_{\text{eq}} - \mathcal{T}(t)}{\mathcal{T}(t)}\,, \\ \alpha_P(t) &= 1 - \frac{\kappa_T \Delta t}{\tau_P}(P_{\text{eq}} - \mathcal{P}(t))\,,\end{aligned} \tag{2}$$

as derived in the lecture [3]. $\tau_{T,P}$ are the decay time constants, which we set to be $300\Delta t$, and κ_T is the isothermal compressibility, which for aluminum we set to 2.21877 Å^3/eV. $\mathcal{T}(t)$ and $\mathcal{P}(t)$ is the instantaneous temperature and pressure respectively. The instantaneous temperature is given by

$$\mathcal{T} = E_{kin}\frac{2}{3Nk_B}, \tag{3}$$

where E_{kin} is the kinetic energy of all N particles. The instantaneous pressure in turn is obtained through the use of the supplied `get_virial_Al` function which returns

$$W = \langle \mathcal{W}(t)\rangle_{\text{time}} = \left\langle \frac{1}{3}\sum_{i=1}^{N} r_i(t)\cdot \boldsymbol{F}_i(t)\right\rangle_{\text{time}} . \tag{4}$$

This yields

$$\mathcal{P} = \frac{Nk_B\mathcal{T} + W}{(N_c a_0)^3} \tag{5}$$

where $N_c = 4$ is the number of supercells in a given direction.

We first perform the simulation with the equilibration process for a solid at $T_{eq} =$ 773 K[1] and $P_{eq} = 1$ bar, where we stop the rescaling after half of the runtime, at 10 ps. The temperature, pressure, lattice constant and particle position for arbitrarily selected particles can be seen in fig. 4 and fig. 5. We can see that both temperature and pressure equilibrate quickly, however, the pressure shows large fluctuations and can even become negative. The average of the instantaneous temperature after equilibration is 774 ± 30 K ($\pm 1\sigma$), while being 1 K higher than T_{eq}, this is still very close. The average of the instantaneous pressure is -20±1200 bar, which represents a large variation in comparison to $P_{eq} = 1$ bar. However, since aluminium in both its liquid and solid state is incompressible, this is not much of an issue, especially because they are in the correct state of matter, and the properties we are interested in mainly depend on temperature.

The lattice constant also quickly approaches its final value $a_0 = 4.09$ Å. It is reasonable that a_0 is increasing since we are decreasing the pressure. We can see from the trajectories of select particles that they fluctuate around a constant mean position.

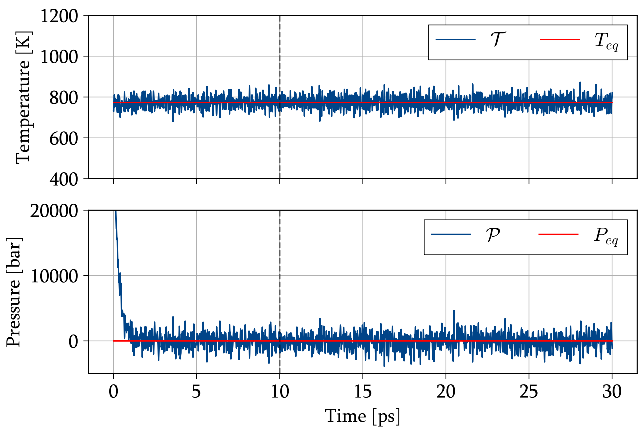

Figure 4: Temperature and pressure for a simulation at $P = 1$ bar and $T = 773$ °C. For the first 10 ps (indicated by dashed line), position and velocity rescaling according toeq. (1) and eq. (2) is performed. The system reaches equilibrium exponentially, whereafter temperature and pressure fluctuate. The fluctuations in pressure are much larger however.

[1]The melting temperature for aluminum is 933 K

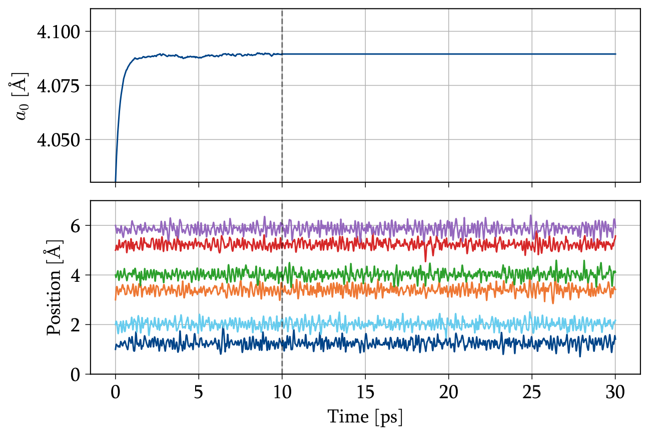

Figure 5: Lattice constant and particle positions for selected particles for a simulation at $P = 1$ bar and $T = 773\,°\text{C}$. The initial positions of the particles are set to $= 1, 2, \ldots, 6$ Å to make it easier to display. The dashed line indicates the end of the equilibration. The lattice constant reaches its equilibrium value exponentially. The system is in a solid state, as is clear from the trajectories which fluctuate around their starting positions.

Problem 4

We repeat the simulation for liquid aluminum, which we heat to 1400 K during the first 10 ps of the equilibration phase in order to melt the lattice faster. We then set its final temperature of $T_{eq} = 973$ K, 40 K above the melting point, during the second part of the equilibration phase, 10 to 20 ps. This is reflected in the temperature curve in fig. 6 aswell as the curve for the lattice constant in fig. 7. The lattice constant is first increasing since we are decreasing the pressure and increasing the temperature. After the inital melting phase it decreases when we lower the temperature. It finally settles at $a_0 = 4.25$ Å. We reach an average temperature $T = 970(34)$ K and pressure $P = -700(2000)$ bar during production. When looking at the particle positions in fig. 7, we can clearly see that the particles undergo diffusion, i.e. the aluminum is liquid, as their trajectories deviate significantly from their starting position.

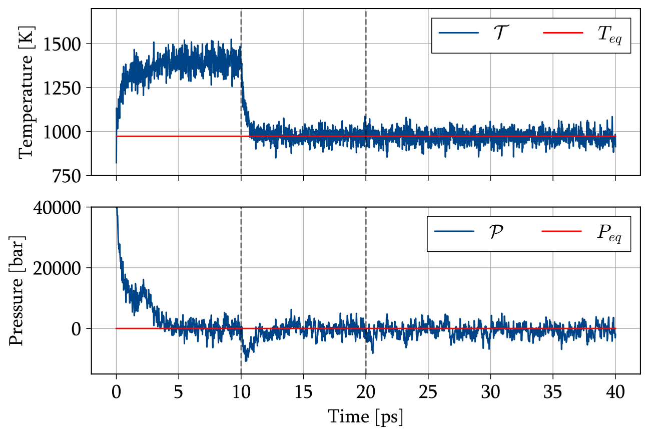

Figure 6: Temperature and pressure for a simulation at $P = 1$ bar and $T = 9730$ K. For the first 10 ps (first dashed line), the system is melted at $T = 1400$ K. Between 10-20 ps (second dashed line), the system is equilibrated to $T = 973$ K. Rescaling is stopped during the last 20 ps. Again, pressure fluctuations are quite large.

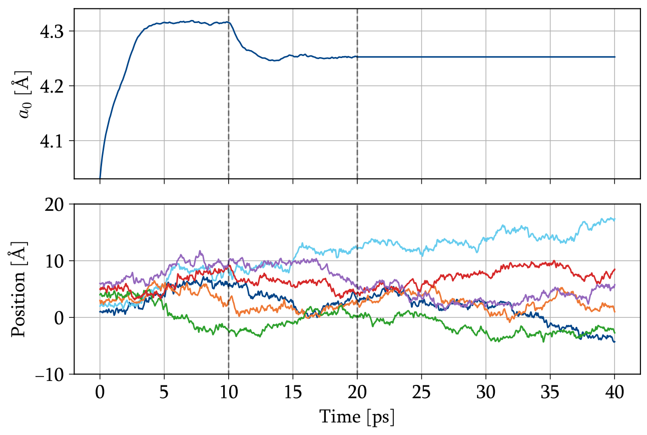

Figure 7: Simulation parameters as in fig. 6. From the trajectories, it is clear that the system is in a liquid state, as the particles undergo diffusion.

Problem 5

We calculate the mean square displacement Δ_{MSD} using

$$\Delta_{\text{MSD},l} = \sum_{i=1}^{N} \sum_{m=0}^{M-l-1} \frac{1}{N(M-l)} (r^i_{m+l} - r^i_m)^2 \,, \tag{6}$$

where the sum over i averages over all particles and spatial components and the sum over m averages over $M - l$ and we consider only every tenth of the timesteps used for simulation (see code in appendix A.3.1) [3]. We use $M = 2000$ and let l run to 10% of M which corresponds to $0.1 \cdot 2000 \cdot \Delta t \cdot 10 = 20$ ps.

For the solid, Δ_{MSD} stays roughly constant after an initial increase, due to the fact that each particle oscillates around its centered, constant position in the lattice. For the liquid, the particles undergo diffusion, leading to a linear increase in the mean square displacement. This is given by

$$\Delta_{\text{MSD}} = 2dD_s t \tag{7}$$

where d is the number of dimensions the particle can move in [3]. We fit a linear curve to obtain a self diffusion coefficient of $D_s = 0.41$ Å^2/s (see code in appendix A.3.2). This can be compared to an experimental value at 980 K of $D_s = 0.72$ Å^2/s [1].

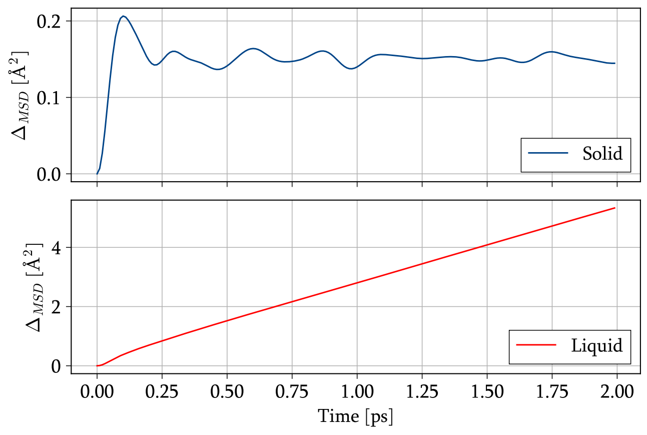

Figure 8: Mean square displacement, calculated from eq. (6), for both the liquid (red) and the solid (blue) system. We average over all particle trajectories. As expected, the Mean square displacement stays constant for the solid, while it increases linearly in time for the liquid. Note the difference in scale on the y-axis.

Problem 6

Now we turn to the velocity autocorrelation function [3]. To do so, we take a similar approach as in calculating the mean square displacement, but consider now instead the product of velocities at different times, i.e. (see code appendix A.4.1)

$$\Phi_l = \sum_{i=1}^{N} \sum_{m=0}^{M-l-1} \frac{1}{N(M-l)} v^i_{m+l} v^i_m \,. \tag{8}$$

This function normalized by its value at $l = 0$ is shown for both the solid (blue) and liquid (red) in fig. 9. For the liquid, the velocity autocorrelation function decays quickly, as is expected for a liquid. For the solid, we can make out a dip, followed by another peak, which follows from the oscillatory motion in the solid state. This correlation vanishes however, when the different oscillators become out of phase due to noise. Moreover, we know from the MD lecture notes that

$$\Phi(0) = \frac{3k_\mathrm{B}T}{m} \tag{9}$$

for a system in equilibrium, which is the case for both our solid an liquid. For the solid and liquid with $T = 773$ K and $T = 973$ K we have that $\frac{3k_\mathrm{B}T}{m}$ is 71.3 and 89.8 Å^2/s^2 respectively (see code appendix A.4.2). Our calculated values for $\Phi(0)$ yields us 71.1 and 89.9 Å^2/s^2 for the solid and liquid respectively. We see that the values match with a high accuracy which indicates that our approach should be valid.

We also consider the cos-transform of the velocity autocorrelation function,

$$\hat{\Phi}(\omega) = 2\int_0^\infty \Phi(t)\cos(\omega t) dt \,. \tag{10}$$

We integrate this using the trapezoidal rule (see code appendix A.4.3) for 10000 linearly spaced values of $\omega \in [0, 100]$ ps^{-1} and obtain the plot in fig. 10. For the solid, we can see peaks that correspond to the harmonic oscillations and their harmonics of the system seen in fig. 9. For the liquid, $\hat{\Phi}(\omega)$ is peaked at the first and largest dip found in fig. 9 after which it decays.

From $\hat{\Phi}$ we can obtain, again, the self diffusion coefficient as

$$D_s = \frac{1}{2d}\hat{\Phi}(0), \tag{11}$$

which yields us $D_s = 0.44$ Å^2/ps (see code appendix A.4.4). This is in quite close agreement with the value of 0.41 Å^2/ps obtained in Problem 5.

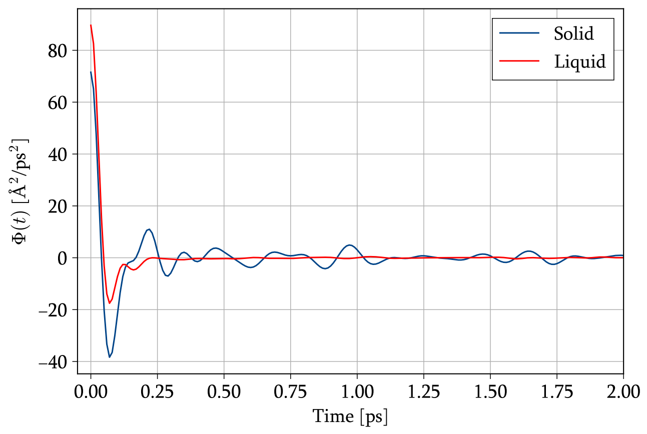

Figure 9: Velocity autocorrelation function for both the liquid (red) and solid (blue) system. The solid curve features a much more pronounced dip after the first peak. This is due to the oscillatory motion in the lattice. Eventually, both correlation functions decay to 0 due to noise.

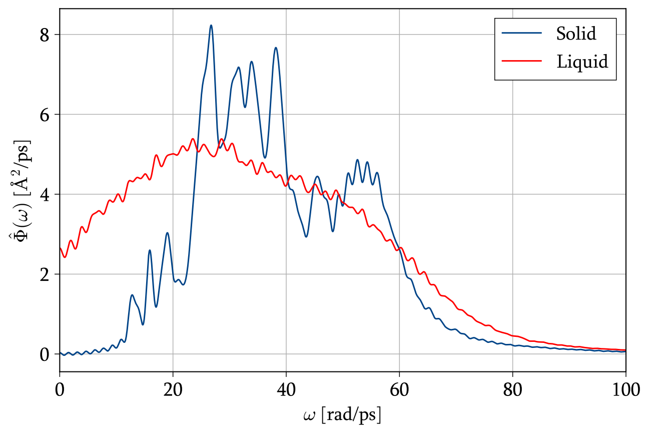

Figure 10: cos transform of the normalized velocity autocorrelation functions shown in fig. 9. The solid shows multiple peaks that correspond to oscillations in the lattice.

Problem 7

We can also obtain the velocity autocorrelation algorithm in a much faster way by using the fast correlation algorithm [3]. Given a velocity for a particle in a certain direction as v_k^i where $i = 1, 2, \dots, 3N$ we add padding to the end of the data accordingly,

$$h_k = \begin{cases} v_k^i; & k = 0, 1, \dots, M-1 \\ 0; & k = M, \dots, N-1 \end{cases} \tag{12}$$

We use $M = 800$ and $N = 2M$ where we have that the distance between two data points is, just as before, $10dt$. We then take the fast fourier transform on h_k to obtain H_k. We then have from the circular correlation theorem that

$$S_l = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \, e^{2\pi i l n/N} \tag{13}$$

where S_l is the correlation function. Its relation to the sought after autocorrelation function is just

$$C_l = \frac{S_l}{M - |l|}. \tag{14}$$

We take this approach on all $3N$ velocity vectors and average the result (see code appendix A.5. The results for the autocorrelation function and the power spectrum obtained through the fast correlation algorithm are shown in fig. 11 and fig. 12 respectively. In comparison to fig. 9, the autocorrelation functions looks identical. Furthermore, $\phi(0) = 71.5$ and 89.6 Å^2/ps^2 for the solid and liquid which is close to both the theoretical values (71.3 and 89.8 Å^2/ps^2) and those obtained in problem 6 (71.1 and 89.9 Å^2/ps^2). This indicated the validity of the method. The main upside with using the fast correlation algorithm to obtain the autocorrelation function is the gain in speed which is many orders of magnitude faster.

For the powerspectrum, fig. 12 is very similar to fig. 9 in the main features. It is however more noisy. For the liquid, we can, just as we did before, obtain the self diffusion coefficient through eq. (11). This yields us a value of $D_s = 0.46$ Å^2/ps which is close to the value of $D_s = 0.44$ Å^2/ps obtained in the previous problem. This too points to the validity of the fast correlation method.

Concluding discussion

We have performed discrete time simulation of a 4x4x4 supercell of fcc Aluminium in both the liquid and the solid state. We have implemented rescaling of positions and velocities to reach equilibrium faster. With this setup, we have recorded the trajectories of the aluminium particles to calculate mean square displacement and velocity autocorrelation functions. We have found different ways to distiguish a solid from a liquid in computer simulations, such as the trajectories of the particles, or certain features in mean square displacement and velocity autocorrelation.

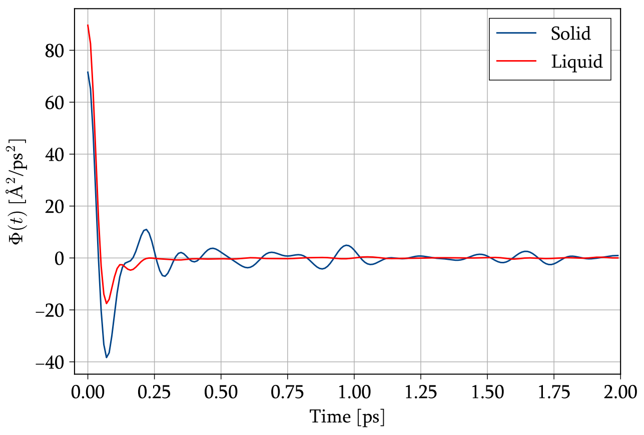

Figure 11: Velocity autocorrelation function obtained through the fast correlation algorithm. Comparable to the autocorrelation function shown in fig. 9.

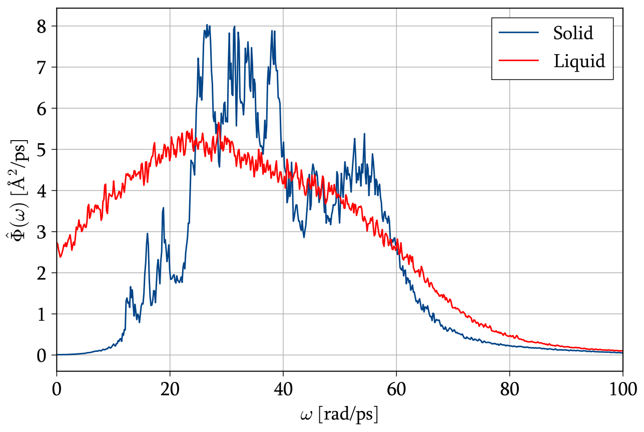

Figure 12: Averaged power spectrum of the velocities of all 256 particles in x, y and z direction. Comparable to the power spectrum shown in fig. 10.

References

[1] F Kargl et al. "Self diffusion in liquid aluminium". In: *Journal of Physics: Conference Series* 340 (Feb. 2012), p. 012077. DOI: 10.1088/1742-6596/340/1/012077. URL: https://doi.org/10.1088/1742-6596/340/1/012077.

[2] *Lattice Constants for all the elements in the Periodic Table*. https://periodictable.com/Properties/A/LatticeConstants.html. (Accessed on 11/22/2020).

[3] Göran Wahnström. *Molecular Dynamics - Lecture notes*. Chalmers university of technology - FKA121. Oct. 2020.

A Source Code

In this appendix we display the source code used in obtaining the results presented in this report. In the first two sections we show the c-code used in problem 1-4. The remaining sections show essential python code used for solving the analysis on the data generated from c.

A.1 Problem 1 - Code

Calculating lattice energy for different lattice parameters

```
    #include <stdio.h>
    #include <math.h>
    #include <stdlib.h>
    #include <time.h>
    #include "H1lattice.h"
    #include "H1potential.h"
    //declare all the variables
    FILE *fp = fopen("H1b_Fig1.csv", "w");
    double a0;
    int Nc = 4;
    double E_pot;
    double pos[4*Nc*Nc*Nc][3];
    int n_data_points=50;
    double a0_max= pow(68,1./3);
    double a0_min = pow(63,1./3);
    //loop over different a0
    for(int i = 0; i<n_data_points; i++){
        a0 = a0_min + (a0_max-a0_min)/n_data_points*i;
        L = a0*Nc;
        init_fcc(pos, Nc, a0);
        E_pot = get_energy_AL(pos, L, N);
        fprintf(fp, "%f,%f\n", a0*a0*a0,E_pot/(pow(Nc,3)));
        }
```

A.2 Problem 2-4 - Code

Implementing verlocity verlet, setting up lattice out of equilibrium, rescaling for faster equilibration, outputting temperature, pressure, lattice size, positions and velocities to file for further analysis using Python.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "H1lattice.h"
#include "H1potential.h"


/* Main program */
int main()
{
    srand(time(NULL));
    //different outputfile for different problems
    //FILE *fp = fopen("H1b_Fig1.csv", "w");
    //FILE *fp2 = fopen("H1b_2.csv", "w");
    FILE *fp3 = fopen("H1b_4.csv", "w");
    FILE *fp4 = fopen("H1b_4_pos.csv", "w");
    int Nc = 4;


    double pos[4*Nc*Nc*Nc][3];
    double v[4*Nc*Nc*Nc][3];
    double forces[4*Nc*Nc*Nc][3];

    //different constants

    double a0 = 4.0318; //use the lattice constant that minimizes potential ↩
        energy at T=0 from prob 1
    double m_Al = 27./9649;
    double E_pot, E_kin;
    double L = a0*Nc;
    int N = 4*Nc*Nc*Nc;
    double dt = 0.001;
    double temperature=0.;
    double pressure;
    int n_timesteps = 30000;
    double alpha_T, alpha_P;
    double kb = 8.617e-5; //Boltzmann constant in eV/K
    double tau_T = 500*dt, tau_P = 500*dt, kappa_T = 2.21877, P_eq = 6.242e-7;
    double  T_eq=1400;
    init_fcc(pos, Nc, a0);
    //fprintf(fp2, "time in ps, E_pot in eV, E_kin in eV, E_tot in eV, ↩
        temperature\n");
   fprintf(fp3, "time in ps, pressure, temperature,pos1x,pos1y,pos1z,pos2x,pos2y↩
       ,pos2z,L\n");

    //add random offsets to positions
    for(int i = 0; i < 4*Nc*Nc*Nc;i++){
        for(int j = 0; j<3; j++){
            pos[i][j]+= (2*((double) rand() / (double) RAND_MAX)-1)*0.065*a0;
            v[i][j]=0;
        }
    }
    //loop over timesteps
    for(int k = 0; k<n_timesteps; k++){

        //begin verlet by calculating the forces on each particle
        get_forces_AL(forces, pos, L, 4*Nc*Nc*Nc);

        for(int i=0; i<4*Nc*Nc*Nc;i++){
            for(int j=0;j<3;j++){
                v[i][j]+= dt * 0.5 * forces[i][j]/m_Al; //translate forces into ↩
                    acceleration F=m*a

            }
        }
        for(int i=0; i<4*Nc*Nc*Nc;i++){
            for(int j=0;j<3;j++){
                pos[i][j]+= dt* v[i][j];

            }
        }
        get_forces_AL(forces, pos, L, 4*Nc*Nc*Nc);
        for(int i=0; i<4*Nc*Nc*Nc;i++){
            for(int j=0;j<3;j++){
                v[i][j]+= dt * 0.5 * forces[i][j]/m_Al;
            }
        }
        E_kin =0;
        for(int i=0; i<4*Nc*Nc*Nc;i++){
            for(int j=0;j<3;j++){
                E_kin+=  0.5 *m_Al* v[i][j]*v[i][j];
            }
        }
        //after one timestep is completed, calculate new energies and temperatur↩
             , pressure
        E_pot =  get_energy_AL(pos, L, N);
        temperature = 2./(3.*N*kb)*E_kin;
        pressure = (N*kb*temperature+get_virial_AL(pos, L, 4*Nc*Nc*Nc))/(pow(L↩
             ,3));
        //output pressure and temperature and unit cell volume  to one file, ↩
             positions to another
        fprintf(fp3, "%f,%f,%f,%f,\n", k*dt,pressure, temperature,pow(L,3));
        //only output the positions and velocities for 1 in 10 timesteps
        if(k%10==0){
        for(int i=0;i<N;i++){
            for(int j = 0; j<3;j++){
            fprintf(fp4, "%f,%f,",pos[i][j],v[i][j]);
            }
        }
        fprintf(fp4, "\n");}


        //rescale positions and velocities for equilibration procedure
        //set target temperature after the Al has melted
        if (k>n_timesteps/4) {
        T_eq = 973;
        }
        //perform the rescaling for the first half of the simulation
        if(k<n_timesteps/2){
        alpha_T = 1 + (2*dt)/(tau_T)*(T_eq-temperature)/temperature; //equations↩
              taken from lecture notes
        alpha_P = 1 - (kappa_T*dt)/(tau_P)*(P_eq-pressure);

        //perform the rescaling
        for(int i=0; i<4*Nc*Nc*Nc;i++){
            for(int j=0;j<3;j++){
                v[i][j] *= pow(alpha_T,0.5);
                pos[i][j]*=pow(alpha_P,1./3);

            }
        }
        //don't forget to rescale cell length!
        L *= pow(alpha_P,1./3);
        }

    }
```

A.3 Problem 5 - Code

A.3.1 Mean squared displacement

Obtain the mean squared displacement function Δ_{MSD} as found in eq. (6).

```python
import numpy as np
def get_delta_MSD(trajectories, M=800, N=256, fraction_of_M=0.2):
    """ Returns the Mean Squared Displacement as a function of l given
    trajectories in x, y and z direction for N particles. l runs from
    0 to M*fraction_of_M."""
    delta_MSD = []
    for l in range(0,int(M*fraction_of_M)): # shift l between trajectories
        temp_sum=0
        for i in range(3*N): # particles in (x,y,z)-direction
            particle = trajectories[:,i]
            for m in range(0, M-l): # run through the data
                temp_sum+=(particle[m+l]-particle[m])**2
        delta_MSD.append(temp_sum*(1/N)*(1/(M-l))) # append and average
    return np.array(delta_MSD)
```

A.3.2 Self diffusion coefficient

Obtain the self diffusion coefficient D_s as found in eq. (7).

```python
import numpy as np
delta_MSD_liquid = get_delta_MSD(...)
D_s = np.polyfit(t, delta_MSD_liquid,deg=1)[0]/6
```

A.4 Problem 6 - Code

A.4.1 Velocity correlation function

Obtain the velocity correlation function Φ as found in eq. (8).

```python
import numpy as np
def get_phi(velocities, M=800, N=256, fraction_of_M=0.2):
    """ Returns the velocity correlation function Phi as a function
    of l given velocities in x, y and z direction for N particles.
    l runs from 0 to M*fraction_of_M."""
    phi = []
    for l in range(0,int(M*fraction_of_M)): # shift l between velocities
        temp_sum=0
        for i in range(N*3): # velocities in (x,y,z)-direction
            particle = velocities[:,i]
            for m in range(0, M-l): # run through the data
                temp_sum+=(particle[m+l]*particle[m])
        phi.append(temp_sum*(1/N)*(1/(M-l))) # append and average
    return np.array(phi)
```

A.4.2 Theoretical value for $\Phi(0)$

Compare theoretical values to calculated values through eq. (9).

```python
def phi_0(T):
    """ Returns the theoretical value of phi(0) for a system in equilibrium at
    temperature T in Celsius for Aluminum"""
    k_b=8.617*10**(-5) # [eV/K] Boltzman constant
    T=T+272.15 # [K] Temperature
    m_al=27.0/9649 # [Atomic units] Mass of aluminium
    phi_0=3*k_b*T/m_al
    return phi_0
```

A.4.3 Spectrum

Calculates the powerspectrum given a velocity correlation function, 10.

```python
import numpy as np
omega=np.linspace(0,100,10000) # inverse ps
def get_phi_hat(phi):
    """ Returns the spectrum phi hat from the velocity correlation function"""
    t=np.linspace(0,len(phi)-1, len(phi))
    return np.array([2*np.trapz(phi*np.cos(o*t),t) for o in omega])
```

A.4.4 Self diffusion coefficient

Obtain the self diffusion coefficient D_s as found in eq. (11).

```
phi_hat_liquid = get_phi_hat(phi_liquid)
self_diffusion_coefficient=1/6*phi_hat_liquid[0]
```

A.5 Problem 7 - Code

Fast correlation algorithm.

```
import numpy as np
def correlation_function_from_power_spectrum(velocities, M=2000, N=256):
    """ Returns the velocity correlation function as a function of l given
    velocities in x, y and z direction for N particles. l runs from 0 to 2*M. ↩
        also returns the powerspectrum, frequencies and time
    """
    padding=np.zeros(M)
    t = np.linspace(0,M*dt*tau*2, M*2 )
    to=3*N
    for i in range(to): #N*3
        h_k = np.concatenate((velocities[:,i],padding))

        power_spectrum = np.abs(np.fft.fft(h_k, n=len(h_k)))**2
        freq = np.fft.fftfreq(t.shape[-1])

        S_l = np.fft.ifft(power_spectrum)
        time = np.fft.fftfreq(freq.shape[-1])
        delta = time[1]-time[0]
        C_l = np.array([S_l[i]/(M-np.abs(l/delta)) for i, l in enumerate(time[0:↩
            M])])
        if i==0:
            C_l_tot=C_l
            PS = power_spectrum
        else:
            C_l_tot+=C_l
            PS+=power_spectrum
    return 3*np.array(C_l_tot)/to, 3*PS/to, freq, time
```