NB: The graded, first version of the report must be returned if you hand in a second time!

H1b: MD simulation – dynamic properties

Andréas Sundström and Linnea Hesslow

November 29, 2018

Task №	Points	Avail. points
$\sum$		

Introduction

The velocity Verlet algorithm is a semi-implicit and efficient method to simulate en ensemble of particles whose trajectories are governed by Newton's equation of motion. Accordingly, it is a suitable algorithm to study molecular dynamics and to determine statistical properties of a system. Here, we use the velocity Verlet algorithm to study a system of aluminum atoms in a face centered cubic (FCC) lattice. By scaling the positions, momenta and lattice parameter, we can equilibrate the temperature and pressure to prescribed values. We study the alumninum system at $500\,^\circ\mathrm{C}$ and $700\,^\circ\mathrm{C}$, which correspond to the solid and liquid state respectively, and compute two dynamic quantities: mean squared displacements and the velocity correlation function.

For this report, we used scripts provided by Anders Lindman for intializing FCC lattices and calcualting lattice energies, lattice forces and virials of an aluminium lattice. All simulations were done in a simulation box containing $4^3 = 64$ unit FCC cells, and a total of $4 \cdot 64$ aluminum atoms placed at the corresponding FCC lattice points. The simulation box had periodic boundary conditions.

The velocity Verlet algorithm

The main idea behind the velocity Verlet algorithm is to split up the time steps of the velocity, so as to make the update process of the state more symmetric. The position, velocity and acceleration, x_i, v_i and a_i[1] respectively, are updated according to

$$\begin{aligned} v_{i+\frac{1}{2}} &= v_i + \frac{1}{2}a_i dt, \\ x_{i+1} &= x_i + v_{i+\frac{1}{2}} dt, \\ a_{i+i} &= \texttt{get\_acceleration}(x_{\imath+1}), \\ v_{i+1} &= v_{i+\frac{1}{2}} + \frac{1}{2}a_{i+1} dt. \end{aligned} \tag{1}$$

By effectively using an average of the old and new acceleration, $(a_{i+1} + a_i)/2$, for updating the velocity, $v_i \to v_{i+1}$, the velocity Verlet algorithm becomes semi-implicit; this also results in better energy-conservation properties of the algorithm, compared to, e.g., a fully explicit algorithm ($v_{i+1} = v_i + a_i dt$). However, in contrast to a fully implicit algorithm, there is no need for a computationally costly matrix inversion for each time step, and the velocity Verlet algorithm is also self-starting on an initial condition of $x_{i=0} = x_0$, $v_{i=0} = v_0$, and $a_{i=0} = \texttt{get\_acceleration}(x_0)$.

Task 1: potential energy

The theoretical, minimum energy lattice parameter for aluminum can be determined by calculating the minimum potential energy per unit cell in a lattice with zero initial momenta for all particles.

Figure 1 shows the potential energy as a function of the lattice parameter. We used a quadratic fit to find the minimum energy[2], and obtained $V_{\mathrm{eq}} \approx 65.38\,\mathrm{Å}^3$. This corresponds to the equilibrium lattice parameter $a_{\mathrm{eq}} \approx 4.029\,\mathrm{Å}$ at $0\,\mathrm{K}$, which we took as the initial lattice parameter for the following tasks. We find that figure 1 looks similar to the figure 1 in the homework problem file, which is encouraging.

Task 2: determine the time step

In this task, we use a lattice with the equilibrium lattice constant $a_{\mathrm{eq}} \approx 4.029\,\mathrm{Å}$, found in the previous task, but then we added a random perturbation, uniformly distributed in the interval $\pm 0.065 a_{\mathrm{eq}}$, to each atom position. This creates a non-equlibrium system, which has a non-trivial time evolution. To determine the time evolution, we used the velocity Verlet algorithm, as described in the introduction.

[1] In most situations the acceleration need not be saved for each time step, which might be insinuated by the index on a_i. The index is just used for notational convenience.

[2] We performed the quadratic fit in the volume V, which to a small error corresponds to a quadratic fit in the lattice parameter a, since $E \approx \alpha(V - V_0)^2 \approx \alpha a_0^4 (a - a_0)^2$ in a close vicinity of the minimum $a \approx a_0$.

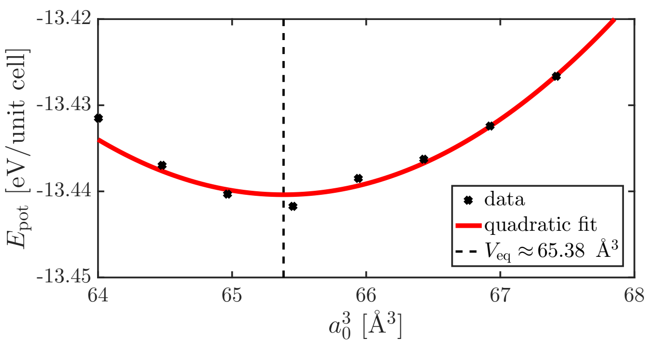

Figure 1: The potential energy per unit cell for aluminum as a function of the lattice parameter cubed.

The first step when doing simulations of this kind, is to determine a suitable timestep. Even though the velocity Verlet algorithm have good energy conservation properties, it still only gives an approximation to the "true" continuous solution; an approximation which gets better the smaller *dt* we choose. However, choosing *dt* too small will result in unnecessary computational costs for the same total simulation time. We are therefore interested in finding the largest *dt* we can get away with without loosing energy conservation. From figure 2, we see that $dt = 2 \cdot 10^{-2}$ ps clearly does not conserve energy, while $dt = 1 \cdot 10^{-1}$ ps dose conserve energy. To be on the safe side, we chose $dt = 5 \cdot 10^{-3}$ ps = 5 fs as our time step. This is in line with the lecture notes, where it is stated that a suitable time step would normally be a few femtoseconds, or somewhat larger for heavy atoms.

The total energy of the simulated system at each time step can easily be calculated as a sum of the kinetic energy of each particle, $E_{\text{kin}}^{(\text{atom})} = m_{\text{Al}}v^2/2$, as well as the total lattice energy of the system. Then, to calculate the temperature, we can use the *equipartition theorem* stating that $\left\langle E_{\text{kin}}^{(\text{atom})} \right\rangle = 3k_{\text{B}}T/2$, or equivalently that $T = 2\left\langle E_{\text{kin}}^{(\text{atom})} \right\rangle/(3k_{\text{B}})$. We can therefore define an instantaneous temperature

$$\mathcal{T}(t) = \sum_{\text{all atoms}} 2E_{\text{kin}}^{(\text{atom})}(t)/(3N_{\text{atoms}}k_{\text{B}}). \tag{2}$$

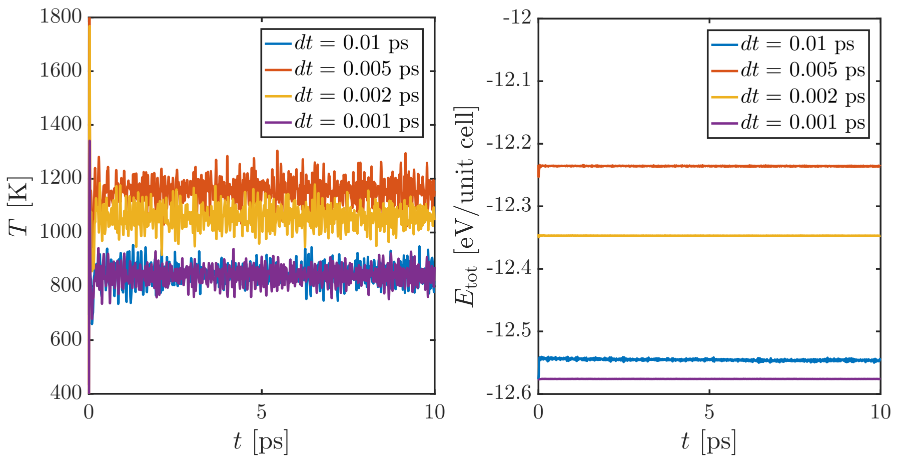

Figure 2: The temperature and kinetic energy per unit cell as a function of time for four different time steps.

With the random noise, the temperature and the energy differ between runs, but are in the same order of magnitude. We note that the temperature in several cases is higher than desired value of 600-800 K from the problem sheet. The temperatures and energies up to one standard deviation are quantified in table 1.

Table 1: NEED UPDATING! Energies and temperatures with one standard deviation uncertainties for four different values of the time steps.

dt [ps]	T [K]	E_{tot} [eV/unit cell]
10^{-2}	$847 \pm 4.2\%$	$-12.55 \pm 1.2 \cdot 10^{-2}\%$
$5 \cdot 10^{-3}$	$1157 \pm 3.8\%$	$-12.24 \pm 3.6 \cdot 10^{-3}\%$
$2 \cdot 10^{-3}$	$1058 \pm 3.7\%$	$-12.35 \pm 6.6 \cdot 10^{-4}\%$
$1 \cdot 10^{-3}$	$841 \pm 3.7\%$	$-12.58 \pm 3.6 \cdot 10^{-4}\%$

Tasks 3 and 4: temperature and pressure equilibration

When we started the system with the random fluctuations, we saw in figure 2 that we get some different temperatures each time. If we want to study the system at some given temperature and pressure, we need some way of persuading the system to change to the desired macro state.

Given that the temperature of the system is given by the average kinetic energy of the atoms, we can change the temperature by scaling the velocities of all atoms. A scheme for this temperature scaling is to first calculate the instantaneous temperature, $\mathcal{T}$ according to equation (2), at that time step, and then scale it by a scaling factor

$$\alpha_T = 1 - \frac{dt}{\tau_T}\frac{\mathcal{T} - T_{\text{eq}}}{\mathcal{T}}, \tag{3}$$

where T_{eq} is the desired equilibrium temperature, and τ_T turns out to be a typical time scale on which the system equlibrates. However, it is only the particle velocities which we have control over, so to actually scale the temperature, we have to scale the velocities by $v_i \to \sqrt{\alpha_T} v_i$, since the temperature depends quadratically on the velocities.

A similar scheme for pressure equlibration is to instead scale the particle positions and simulation-box volume, using a scaling factor of

$$\alpha_P = 1 - \kappa\frac{dt}{\tau_P}(\mathcal{P} - P_{\text{eq}}), \tag{4}$$

where $\mathcal{P}$ is the instantaneous pressure, P_{eq} is the desired equlibrium pressure, τ_P is the characteristic equlibration time, and κ is the isothermal compressibility of the material simulated[3]. This time the positions are scaled according to $x_i \to \alpha_P^{1/3} x_i$, and similarily the simulation box volume is scale by scaling its side lengths by $L \to \alpha_P^{1/3} L$.

We set $\tau_P = \tau_T = 100dt$, where $dt = 5 \cdot 10^{-3}$ ps, and equilibrated the temperature and pressure by scaling the particle momenta and positions (and box size) respectively. Choosing a slower equilibration time did not affect the results qualitatively. Both temperature and pressure were equilibrated in the same Verlet loop, but for the higher temperature the system was first melted by increasing the temperature to 1100 °C. To determine the isothermal compressibility κ, the values of Young's modulus Y and shear modulus G were taken from Physics Handbook, table T 1.1. From F 1.15 in Physics Handbook, the bulk modulus can then calculated as

$$B = \frac{YG}{9G - 3Y} \qquad \kappa_{\text{Al}} = \frac{1}{B} \approx \left(6.6444 \cdot 10^5 \text{ bar}\right)^{-1}, \tag{5}$$

where $1 \text{ bar } = 6.2415 \cdot 10^{-7}$ eV/Å^3 in atomic units.

The results are shown in figure 3, where we overlay the instantaneous values of $\mathcal{T}$ and $\mathcal{P}$ with a moving average using 250 time steps. The desired temperatures and pressures were approximately obtained in the equilibration process. Although the average pressure was slightly below zero, this is within the fluctuation error bars and is in line with the figure 2 in the homework problem document. The equilibrium values of the lattice parameter were found to be

$$a_0 \approx 4.09 \text{ Å}, \quad T = 500 \,°\text{C}, \tag{6}$$

$$a_0 \approx 4.25 \text{ Å}, \quad T = 700 \,°\text{C}. \tag{7}$$

These value are larger than the zero-temperature constants, and it is reasonable that the higher 700 °C case corresponds to a larger lattice parameter at constant pressure.

[3]The isothermal compressibility is defined according to $\kappa = -(V\ \partial P/\partial V)_T^{-1}$. This compressibility should therefore be used when scaling the volume of the box to change the pressure.

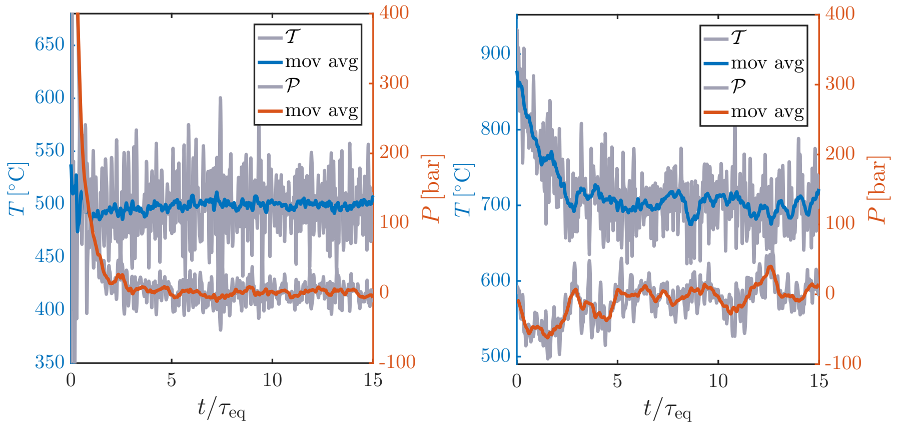

Figure 3: The instantaneous values of $\mathcal{T}$ and $\mathcal{P}$ overlaid with with a moving average using 100 time steps, which corresponds to $\Delta t = \tau_P/2$. Left panel: $T = 500\,^\circ\mathrm{C}$, right panel: $T = 700\,^\circ\mathrm{C}$.

Tasks 3-5: particle trajectories

Starting with the temperature- and pressure-equilibrated systems from the previous section, we study the particle trajectories for both systems. Here, we decrease the time step to $dt = 5 \cdot 10^{-4}$ ps and the simulation length to $t_{\mathrm{end}} = 5$ ps to get better statistics. This was mostly motivated by increasing the resolution in tasks 6-7.

First, we note that the cumulative averages of the instantaneous temperatures and pressures stayed close to their initial values. This is shown in figure 4.

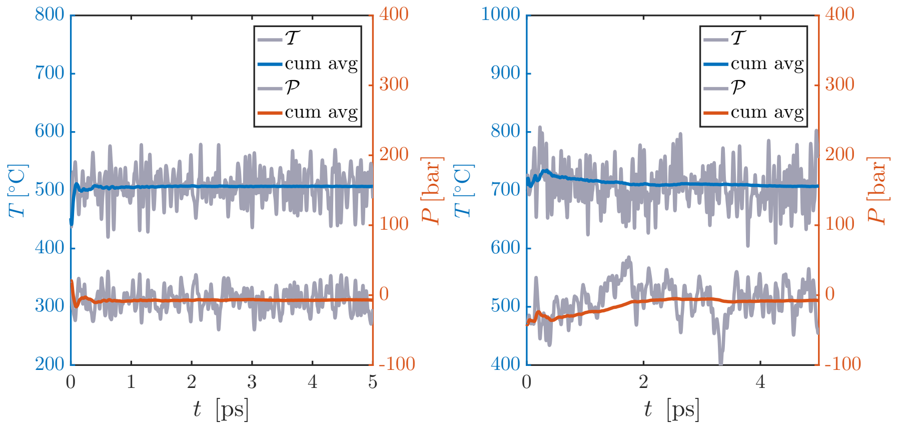

Figure 4: The instantaneous values, and the cumulative averages, of the temperature and the pressure in the production runs. Left panel: $T = 500\,^\circ\mathrm{C}$, right panel: $T = 700\,^\circ\mathrm{C}$

From equation (82) in MD lecture notes, the mean squared displacement can be calculated as

$$\Delta_{\mathrm{MSD}}(t) = \lim_{T\to\infty} \frac{1}{T} \int_0^T dt' \frac{1}{N_{\mathrm{atoms}}} \sum_{i=0}^{N_{\mathrm{atoms}}-1} \left[\mathbf{r}_i(t+t') - \mathbf{r}_i(t')\right]^2 \tag{8}$$

$$\Rightarrow$$

$$\Delta_{\mathrm{MSD}}(t_k) \approx \frac{1}{N_T - k} \frac{1}{N_{\mathrm{atoms}}} \sum_{j=0}^{N_T-k-1} \sum_{i=0}^{N_{\mathrm{atoms}}-1} \left[\mathbf{r}_i(t_{k+j}) - \mathbf{r}_i(t_j)\right]^2 \tag{9}$$

We now consider the particle trajectories. Figure 5 shows the trajectories of five individual particles along with the mean squared displacement as determined in equation (9). We can clearly see that the particle trajectories are bounded in the left figure 5, while for the high-temperature case in the right panel, they increase as square root of time ($\Delta_{\text{MSD}} \propto t$). Consequently, the former is in a solid state while the latter is in a liquid state.

The self-diffusion coefficient as determined by the average slope of the mean squared displacement, was calculated to $D_s \approx 0.52\,\text{Å}^2/\text{ps}$.

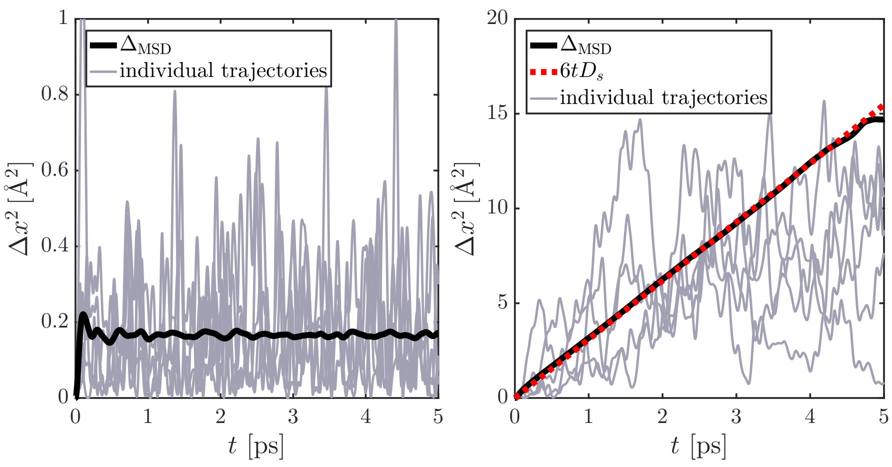

Figure 5: Five individual particle trajectories (gray thin lines), overlaid with the mean squared displacement (thick black line). In the left panel, $T = 500\,°\text{C}$, the system is in a solid state. In the right panel, $T = 700\,°\text{C}$, the system is in a liquid state, where $\Delta_{\text{MSD}} \approx 6tD_s$ (dotted red line).

Tasks 6-7: velocity correlation and power spectrum

We calculated the discrete auto-correlation function similarly to the MSD,

$$\Phi_j = \frac{1}{N-j} \sum_{i=0}^{N-j-1} \left\langle v_{i+j} v_i \right\rangle, \tag{10}$$

where $j = 0, 1, \ldots, N-1$ and the average is taken over all atoms. Figure 6(left) shows that it is noticeably different between the solid and the liquid states: while the solid state remains non-zero at longer time, presumably because of oscillations around lattice points, the liquid velocity correlation quickly decays to zero after some initial oscillations.

We then preceded to numerically approximate the integral

$$\hat{\Phi}(f) = 2\int_0^{\infty} \mathrm{d}t\ \Phi(t)\cos(2\pi f t) \approx 2\int_0^{T_\text{s}} \mathrm{d}t\ \Phi(t)\cos(2\pi f t) \tag{11}$$

using a trapezoidal method in Matlab, with a frequency range $f = 0$ to $f = 1/(2\Delta t) = f_{\text{Nyqvist}}$, and frequency steps $\Delta f = 1/T_\text{s}$, where T_s is a time at about half the simulation end time. This is to avoid including noisy data in $\Phi(t)$ at later times, where the statistics are poor.

We then calculated the power spectrum according to

$$\begin{aligned} \hat{P}(\omega) &= \lim_{T\to\infty} \frac{1}{T} \left\langle \left| \int_0^T \mathrm{d}t\ v(t) e^{\mathrm{i}\omega t} \right|^2 \right\rangle \\ &\approx \frac{1}{T} \left\langle \left| \int_0^T \mathrm{d}t\ v(t) e^{\mathrm{i}\omega t} \right|^2 \right\rangle \\ \Rightarrow \quad \hat{P}_k &= \frac{1}{T} \left\langle \left| \frac{T}{N} \sum_{i=0}^{N-1} v_i \exp\left(\mathrm{i}2\pi \frac{ik}{N}\right) \right|^2 \right\rangle = \frac{T}{N} \left\langle |\hat{v}_k|^2 \right\rangle \end{aligned} \tag{12}$$

where the averages is taken over all atoms, and

$$\hat{v}_k = \sqrt{N} \sum_{i=0}^{N-1} v_i \exp\left(\mathrm{i}2\pi \frac{ik}{N}\right) \tag{13}$$

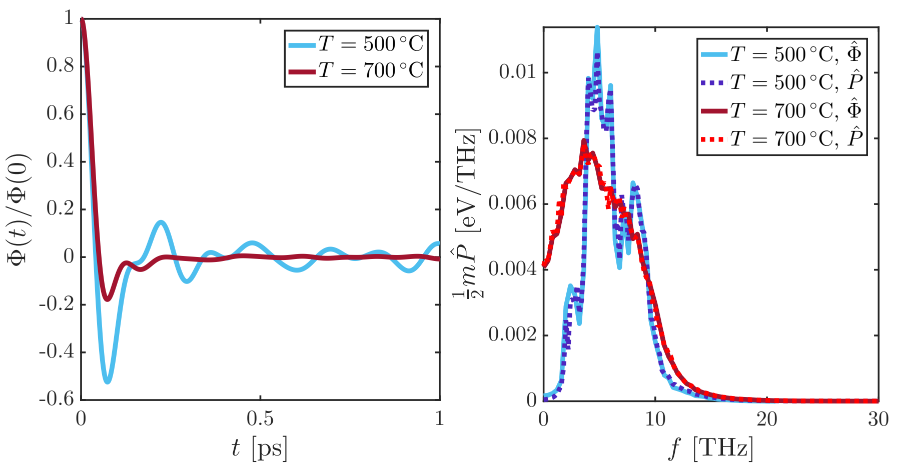

Figure 6: Left panel: The velocity correlation function, and (right panel) its spectrum, calculated both directly from the velocity correlation (solid line) and from the power spectrum of the particle velocity (dotted line). Blue lines show $T = 500\,^\circ$C and red lines $T = 700\,^\circ$C. The spectrum is multiplied by a factor of $\frac{1}{2}m_{\mathrm{Al}}$, in which case it can be interpreted as energy per frequency interval and atom.

is the discrete Fourier transform of v_i. Note that the factor of T was not included in the C scripts, and we therefore multiplied by this factor in the Matlab plotting scripts. When we compare $\hat{\Phi}_k$ and $\hat{P}_k$ in figure 6 (right), we find that they are very similar, as, indeed, they should be according to the Wiener-Khinthchine theorem. If we instead take $T_s = T$ (using the full time evolution of $\Phi(t)$, we get a more noisy signal. This is because there is less statistics at high values of j in equation (10). Therefore, even though the results are similar, the power spectrum method in equation (12), should be considered more accurate since it includes more statistics of the data points.

The self-diffusion coefficient as determined by the power spectral density at $f = 0$, was found to be $D_s = 0.49\,\text{Å}^2/\text{ps}$, which is close to the value obtained from the mean squared displacement, as expected.

Concluding discussion

Using the velocity Verlet algorithm, we study a system of aluminum atoms at 500° C and 700° C, which correspond to the solid and liquid state respectively.

From both the mean squared displacements and the velocity correlation function, the solid state is clearly distinguishable from the liquid state. The mean squared displacement reaches a constant value in the solid state, whereas it grows linearly with time in the liquid state, which is characteristic of diffusion in a random walk process. Similarly, the spectrum of the velocity correlation function vanishes at zero frequency which means that the average velocity correlation is zero and hence there is no net movement of the particles; in contrast for the liquid state, the zero-frequency value of the spectrum is finite and proportional to the diffusion coefficient.

A Source Code

A.1 Main program task 1: `main_T1.c`

```c
/*
  main_T1.c Task 1 H1b
  In this task, we scan over a range of lattice parameters, a0, to determine
  which one results in the lowest potential energy stored in the lattice.

  System of units:
  Energy   - eV
  Time     - ps
  Length   - Angstrom
  Temp     - K
  Mass     - eV (ps)^2 A^(-2)
  Pressure - eV A^(-3)
*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#include "initfcc.h"
#include "alpotential.h"

#define N_cells 4
#define N_lattice_params 25

/* Main program */
int main()
{
  int N_atoms = 4*N_cells*N_cells*N_cells;
  double a0;
  double a0_min = 4.0;
  double a0_max = 4.2;
  double da0 = (a0_max - a0_min)/N_lattice_params;

  double (*pos)[3] = malloc(sizeof(double[N_atoms][3]));
  double *energy = malloc(sizeof(double[N_lattice_params]));

  FILE *file_pointer;

  for (int i=0; i<N_lattice_params; i++){
    a0 = a0_min + i*da0;         // The lattice constant of this iteration
    init_fcc(pos, N_cells, a0); // Init, FCC cells with lattice constant `a0`
    // energy per unit cell
    energy[i] = get_energy_AL(pos, N_cells*a0, N_atoms )*4/N_atoms;
  }

  // Write to files
  file_pointer = fopen("../data/lattice_energies.tsv", "w");
  for (int i=0; i<N_lattice_params; i++){
    a0 = a0_min + i*da0;
    fprintf(file_pointer, "%.8f \t %.8f \n", a0, energy[i]);
  }
  fclose(file_pointer);

  free(pos);    pos = NULL;
  free(energy); energy = NULL;
  return 0;
}
```

A.2 Main program Task 2: `main_T2.c`

```c
/*
  main_T2.c, Task 2, H1b
  In this task, we add random noise to the particle positions and see how the
  system evolves in time. Using the kinetic energy of the particles, we can
  derive an instantaneous temperature of the system.

  System of units:
  Energy   - eV
  Time     - ps
  Length   - Angstrom
  Temp     - K
  Mass     - eV (ps)^2 A^(-2)
  Pressure - eV A^(-3)
*/

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

```

```c
#include "initfcc.h"
#include "alpotential.h"
#include "funcs.h"

#define N_cells 4
#define AMU 1.0364e-4
#define kB 8.6173303e-5

/* Main program */
int main()
{
  int N_atoms = 4*N_cells*N_cells*N_cells;
  double m_Al = 27*AMU;

  double a_eq = 4.03; // Min potential energy lattice constant

  double noise_amplitude = 6.5e-2 * a_eq;
  double t_max=10; //
  double dt = 1e-3;
  int N_timesteps = t_max/dt;
  double t, E_kin;

  double (*pos)[3] = malloc(sizeof(double[N_atoms][3]));
  double (*momentum)[3] = malloc(sizeof(double[N_atoms][3]));
  double (*forces)[3] = malloc(sizeof(double[N_atoms][3]));
  double *temperature = malloc(sizeof(double[N_timesteps]));
  double *E_tot = malloc(sizeof(double[N_timesteps]));

  FILE *file_pointer;

  init_fcc(pos, N_cells, a_eq); // initialize fcc lattice
  add_noise( N_atoms, 3, pos, noise_amplitude ); // adds random noise to pos
  set_zero( N_atoms, 3, momentum); // set momentum to 0
  get_forces_AL( forces, pos, a_eq*N_cells, N_atoms); //initial cond forces

  for (int i=0; i<N_timesteps; i++){
    /*
       The loop over the timesteps first takes a timestep according to the
       Verlet algorithm, then calculates the energies and temeperature.
    */
    timestep_Verlet (N_atoms, pos,  momentum, forces, m_Al, dt, a_eq*N_cells);

    E_kin    = get_kin_energy(N_atoms, momentum, m_Al );
    E_tot[i] = (E_kin + get_energy_AL(pos, a_eq*N_cells, N_atoms))*4/N_atoms;

    /* 3N*kB*T/2 = 1/(2m) * \sum_{i=1}^{N} p_i^2  = p_sq/(2m) */
    temperature[i] =  E_kin * 2/(3*N_atoms*kB);
  }

  /* Write tempertaure to file */
  char file_name[100];
  sprintf(file_name,"../data/temperature_dt-%0.0e_Task2.tsv", dt);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_timesteps; i++){
    t = i*dt; // time at step i
    fprintf(file_pointer, "%.4f \t %.8f \n", t, temperature[i]);
  }
  fclose(file_pointer);

  /* Write total energy to file */
  sprintf(file_name,"../data/total_energy_dt-%0.0e_Task2.tsv", dt);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_timesteps; i++){
    t = i*dt; // time at step i
    fprintf(file_pointer, "%.4f \t %.8f \n", t, E_tot[i]);
  }
  fclose(file_pointer);

  free(pos);         pos = NULL;
  free(momentum);    momentum = NULL;
  free(forces);      forces = NULL;
  free(temperature); temperature = NULL;
  free(E_tot);       E_tot = NULL;
  return 0;
}
```

A.3 Temperature and pressure equilibration for tasks 3-7: main_T3.c

```c
/*
  main_T3.c, Tasks 3 and 4, H1b. Also used as input in Tasks 5-7.
  In this task, we use an equlibration scheme, based on scaling particle momenta
  and positions, to equlibrate the temperature and pressure in the system. We do
  this for T=500 degC and T=700 degC and P=1 bar. The difference between the two
  temperatures are that the higer temperature results in a melted system. (To
  ensure that the system is melted properly, we first raise the temperature to
  900 degC and then lower it back to 700 degC.)

  After the system has equlibrated, we save the full phase space (all particle
  positions and momenta) as well as the equlibrated lattice parameter to a
  binary file which then can be read in for a production run.

  System of units:
  Energy   - eV
  Time     - ps
  Length   - Angstrom
  Temp     - K
  Mass     - eV (ps)^2 A^(-2)
  Pressure - eV A^(-3)
*/

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

#include "initfcc.h"
#include "alpotential.h"
#include "funcs.h"

#define N_cells 4
/* define constants in atomic units: eV,    , ps, K */
#define AMU 1.0364e-4
#define degC_to_K 273.15
#define bar 6.2415e-07
#define kB 8.61733e-5

/* Main program */
int main()
{
  char file_name[100];

  int N_atoms = 4*N_cells*N_cells*N_cells;
  double m_Al = 27*AMU;
  /*
    Values of Young's and shear modulus, Y and G resp., taken from
    Physics Handbook, table T 1.1. Bulk mudulus then calculated as
    B = Y*G / (9*G - 3*Y)   [F 1.15, Physics Handbook]
    kappa = 1/B
  */
  double kappa_Al = 100/(6.6444e+05 * bar); // STRANGE FACTOR 100 OFF !!!
  double a_eq = 4.03;
  double cell_length = a_eq*N_cells;
  double inv_volume = pow(N_cells*cell_length, -3);
  double noise_amplitude = 6.5e-2 * a_eq;

  double T_final_C= 500;
  int nRuns = 1; //2 if melt, 1 otherwise
  double T_melt_C = 900;

  double P_final_bar= 1;

  double T_eq;
  double P_eq  = P_final_bar*bar;
  double dt    = 5e-3;
  double tau_T = 100*dt;
  double tau_P = 100*dt;
  double t_eq= 15*tau_P; //equlibration times
  int N_timesteps = t_eq/dt;

  double alpha_T, alpha_P,alpha_P_cube_root;
  double t, E_kin, virial;

  double (*pos)[3] = malloc(sizeof(double[N_atoms][3]));
  double (*momentum)[3] = malloc(sizeof(double[N_atoms][3]));
  double (*forces)[3] = malloc(sizeof(double[N_atoms][3]));
  double *temperature = malloc(sizeof(double[N_timesteps]));
  double *pressure = malloc(sizeof(double[N_timesteps]));

  FILE *file_pointer;


  init_fcc(pos, N_cells, a_eq); // initialize fcc lattice
  add_noise( N_atoms, 3, pos, noise_amplitude ); // adds random noise to pos
  set_zero( N_atoms, 3, momentum); // set momentum to 0
  get_forces_AL( forces, pos, cell_length, N_atoms); //initial cond forces

  for (int irun=0; irun < nRuns; irun++){// last run: final, irun = 0
    if (irun == nRuns - 1){ // final run
      T_eq = T_final_C + degC_to_K;
    }else{
      T_eq = T_melt_C + degC_to_K;
    }
    for (int i=0; i<N_timesteps; i++){
      /*
         The loop over the timesteps first takes a timestep according to the
```

```
       Verlet algorithm, then calculates the energies and temeperature.
    */
    timestep_Verlet(N_atoms, pos,  momentum, forces, m_Al, dt, cell_length);

    E_kin  = get_kin_energy(N_atoms, momentum, m_Al );
    virial = get_virial_AL(pos, cell_length, N_atoms);

    /* 3N*kB*T/2 = 1/(2m) * \sum_{i=1}^{N} p_i^2  = p_sq/(2m) */
    temperature[i] =  E_kin * 1/(1.5*N_atoms*kB);
    /* PV = NkT + virial */
    pressure[i] = inv_volume * (1.5*E_kin + virial);

    /* Equlibrate temperature by scaling momentum by a factor sqrt(alpha_T).
        N.B. It is equally valid to scale the momentum instead of the velocity↩
       ,
        since they only differ by a constant factor m.
    */
    alpha_T = 1 + 2*dt*(T_eq - temperature[i]) / (tau_T * temperature[i]);
    scale_mat(N_atoms, 3, momentum, sqrt(alpha_T));

    // Equlibrate pressure by scaling the posistions by a factor of
    // alpha_P^(1/3)
    alpha_P = 1 - kappa_Al* dt*(P_eq - pressure[i])/tau_P;
    alpha_P_cube_root = pow(alpha_P, 1.0/3.0);
    scale_mat(N_atoms, 3, pos, alpha_P_cube_root);

    cell_length*=alpha_P_cube_root;
    inv_volume*=1/alpha_P;

    temperature[i]*=alpha_T;
    pressure[i]*=alpha_P;
   }
  }

  printf("equilibrium a0 = %.4f A\n", cell_length/N_cells);

  /* Write tempertaure to file */
  sprintf(file_name,"../data/temp-%d_pres-%d_Task3.tsv",
      (int) T_final_C, (int) P_final_bar);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_timesteps; i++){
    t = i*dt; // time at step i
    fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n",
        t, temperature[i],pressure[i]);
  }
  fclose(file_pointer);

  /* Write phase space coordinates to file */
  sprintf(file_name,"../data/phase-space_temp-%d_pres-%d.tsv",
      (int) T_final_C, (int) P_final_bar);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_atoms; i++){
    for (int j=0;j<3;j++){
      fprintf(file_pointer, " %.16e \t", pos[i][j]);
    }
    for (int j=0;j<3;j++){
      fprintf(file_pointer, " %.16e \t", momentum[i][j]);
    }
    fprintf(file_pointer,"\n");
  }
  fclose(file_pointer);

  /* save equlibrated position and momentum as a binary file */
  sprintf(file_name,"../data/INIDATA_temp-%d_pres-%d.bin",
      (int) T_final_C, (int) P_final_bar);
  file_pointer = fopen(file_name, "wb");
  fwrite(pos, sizeof(double), 3*N_atoms, file_pointer);
  fwrite(momentum, sizeof(double), 3*N_atoms, file_pointer);
  fwrite(&cell_length, sizeof(double), 1, file_pointer);
  fclose(file_pointer);

  free(pos); pos = NULL;
  free(momentum); momentum = NULL;
  free(forces); forces = NULL;
  free(temperature); temperature = NULL;
  free(pressure); pressure = NULL;
  return 0;
}
```

A.4 Production runs for tasks 3-7 : main_Prod.c

```
/*
  main_Prod.c, Production runs, H1b
  In this program, we use the equlibrated micro-states from Tasks 3-4 to study
  dynamical properties, such as mean squared displacement (MSD), velocity
  auto-correlation function, and the power spectral density of the atom
  movements.

  System of units:
  Energy   - eV
  Time     - ps
  Length   - Angstrom
  Temp     - K
  Mass     - eV (ps)^2 A^(-2)
  Pressure - eV A^(-3)
*/

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

#include "initfcc.h"
#include "alpotential.h"
#include "funcs.h"

#define N_cells 4
/* define constants in atomic units: eV,   , ps, K */
#define AMU 1.0364e-4
#define degC_to_K 273.15
#define bar 6.2415e-07
#define kB 8.61733e-5

/* Main program */
int main()
{
  char file_name[100];

  int N_atoms = 4*N_cells*N_cells*N_cells;
  double m_Al = 27*AMU;
  double cell_length;
  double inv_volume;

  double T_eq_C   = 500;
  double P_eq_bar = 1;

  double dt       = 5e-4; // higher res for spectral function
  double t_end    = 5;
  int N_timesteps = t_end/dt;
  int N_between_steps = 1; // save all steps for max res in spectral function
  int N_save_timesteps = N_timesteps / N_between_steps;
  int N_save_atoms = 5;

  double t, E_kin, virial;

  double (*pos)[3]      = malloc(sizeof(double[N_atoms][3]));
  double (*pos_0)[3]    = malloc(sizeof(double[N_atoms][3]));//for displacements
  double (*momentum)[3] = malloc(sizeof(double[N_atoms][3]));
  double (*forces)[3]   = malloc(sizeof(double[N_atoms][3]));
  double (*displacements)[N_save_atoms] =
    malloc(sizeof(double[N_save_timesteps][N_save_atoms]));
  double (*pos_all)[N_atoms][3] =
    malloc(sizeof(double[N_save_timesteps][N_atoms][3]));
  double (*vel_all)[N_atoms][3] =
    malloc(sizeof(double[N_save_timesteps][N_atoms][3]));
  double *temperature   = malloc(sizeof(double[N_timesteps]));
  double *pressure      = malloc(sizeof(double[N_timesteps]));
  double *msd           = malloc(sizeof(double[N_save_timesteps]));
  double *vel_corr      = malloc(sizeof(double[N_save_timesteps]));
  double *pow_spec      = malloc(sizeof(double[N_save_timesteps]));
  double *freq      = malloc(sizeof(double[N_save_timesteps]));

  // Initialize to 0
  for (int i = 0; i<N_save_timesteps; i++){
    msd[i] = 0;
    pow_spec[i] = 0;
    vel_corr[i] = 0;
  }
  FILE *file_pointer;

  // read positions, momenta and cell_length
  sprintf(file_name,"../data/INIDATA_temp-%d_pres-%d.bin",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "rb");
  fread(pos, sizeof(double), 3*N_atoms, file_pointer);
  fread(momentum, sizeof(double), 3*N_atoms, file_pointer);
  fread(&cell_length, sizeof(double), 1, file_pointer);
  fclose(file_pointer);

  for (int i=0; i<N_atoms; i++){
    for (int j=0; j<3; j++){
      pos_0[i][j]=pos[i][j];
    }
  }
  inv_volume = pow(N_cells*cell_length, -3);
  get_forces_AL( forces, pos, cell_length, N_atoms); //initial cond forces

```

```c
  printf("Initialized. Starting with Verlet timestepping.\n");
  for (int i=0; i<N_timesteps; i++){
    /*
       The loop over the timesteps first takes a timestep according to the
       Verlet algorithm, then calculates the energies and temeperature.
    */
    timestep_Verlet(N_atoms, pos,  momentum, forces, m_Al, dt, cell_length);

    E_kin  = get_kin_energy(N_atoms, momentum, m_Al );
    virial = get_virial_AL(pos, cell_length, N_atoms);
    /* PV = NkT + virial */
    pressure[i] = inv_volume * (1.5*E_kin + virial);
    /* 3N*kB*T/2 = 1/(2m) * \sum_{i=1}^{N} p_i^2  = p_sq/(2m) */
    temperature[i] =  E_kin * 1/(1.5*N_atoms*kB);

    if (i % N_between_steps == 0){
      int k = i/N_between_steps; // number of saved timesteps so far
      // Saves the displacements of some atoms into `displacements`
      get_displacements (N_save_atoms,  pos, pos_0, displacements[k]);

      // Saves all the positions
      copy_mat(N_atoms, 3, pos, pos_all[k]);

      // Saves all the velocities
      copy_mat(N_atoms, 3, momentum, vel_all[k]);
      //But we need to scale the momenta to get the velocities
      scale_mat(N_atoms, 3, vel_all[k], 1/m_Al);
    }

    if ((i*10) % N_timesteps == 0){ //Print out progress at every 10%
      printf("done %d0%% of Verlet timestepping\n", (i*10)/N_timesteps);
    }
  }
  printf("done 100%% of Verlet timestepping\n");

  //Calculating MSD
  printf("calculating MSD\n");
  get_MSD(N_atoms, N_save_timesteps, pos_all, msd);

  //Calculating the velocity correlation function
  printf("calculating velocity correlation\n");
  get_vel_corr(N_atoms, N_save_timesteps, vel_all, vel_corr);

  //Calculating the velocity power spectrum
  printf("calculating power spectrum\n");
  get_powerspectrum(N_atoms, N_save_timesteps, vel_all, pow_spec);
  fft_freq(freq, dt, N_save_timesteps);

  printf("writing to file\n");
  /* Write tempertaure to file */
  sprintf(file_name,"../data/temp-%d_pres-%d_Prod-test.tsv",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_timesteps; i++){
    t = i*dt; // time at step i
    fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n",
        t, temperature[i],pressure[i]);
  }
  fclose(file_pointer);

  /* Write displacements to file */
  sprintf(file_name,"../data/temp-%d_pres-%d_displacements.tsv",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_save_timesteps; i++){
    t = i*dt*N_between_steps; // time at step i
    fprintf(file_pointer, "%.4f", t);
    for (int j=0; j<N_save_atoms; j++){
      fprintf(file_pointer, "\t %.8f", displacements[i][j]);
    }
    fprintf(file_pointer, "\n");
  }
  fclose(file_pointer);

  /* Write MSD to file */
  sprintf(file_name,"../data/temp-%d_pres-%d_dynamicProperties.tsv",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "w");
  // write header
  fprintf(file_pointer, "%% t[ps] \t MSD[A^2] \t vel_corr [A/ps]^2 \n");
  for (int i=0; i<N_save_timesteps; i++){
    t = i*dt*N_between_steps; // time at step i
    fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n", t, msd[i], vel_corr[i]);
  }
  fclose(file_pointer);

  /* Write power spectrum to file */
  sprintf(file_name,"../data/temp-%d_pres-%d_power-spectrum.tsv",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "w");
  // write header
  fprintf(file_pointer, "%% f[1/ps] \t P[A/ps]^2 \n");
  for (int i=0; i<N_save_timesteps/2; i++){ // only save from f=0 to f_crit
    fprintf(file_pointer, "%.4f \t %.8f \n", freq[i], pow_spec[i]);
  }
  fclose(file_pointer);

  // Freeing all the memory
  free(pos);           pos = NULL;
  free(pos_0);         pos_0 = NULL;
  free(momentum);      momentum = NULL;
  free(forces);        forces = NULL;
  free(temperature);   temperature = NULL;
  free(pressure);      pressure = NULL;
  free(displacements); displacements = NULL;
  free(pos_all);       pos_all = NULL;
  free(vel_all);       vel_all = NULL;
  free(msd);           msd = NULL;
  free(vel_corr);      vel_corr = NULL;
  free(pow_spec);      pow_spec = NULL;
  free(freq);          freq = NULL;

  return 0;
}
```

A.5 Production runs for tasks 3-7 : main_Prod.c

```c
/*
  main_Prod.c, Production runs, H1b
  In this program, we use the equlibrated micro-states from Tasks 3-4 to study
  dynamical properties, such as mean squared displacement (MSD), velocity
  auto-correlation function, and the power spectral density of the atom
  movements.

  System of units:
  Energy   - eV
  Time     - ps
  Length   - Angstrom
  Temp     - K
  Mass     - eV (ps)^2 A^(-2)
  Pressure - eV A^(-3)
*/

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

#include "initfcc.h"
#include "alpotential.h"
#include "funcs.h"

#define N_cells 4
/* define constants in atomic units: eV,   , ps, K */
#define AMU 1.0364e-4
#define degC_to_K 273.15
#define bar 6.2415e-07
#define kB 8.61733e-5

/* Main program */
int main()
{
  char file_name[100];

  int N_atoms = 4*N_cells*N_cells*N_cells;
  double m_Al = 27*AMU;
  double cell_length;
  double inv_volume;

  double T_eq_C   = 500;
  double P_eq_bar = 1;

  double dt       = 5e-4; // higher res for spectral function
  double t_end    = 5;
  int N_timesteps = t_end/dt;
  int N_between_steps = 1; // save all steps for max res in spectral function
  int N_save_timesteps = N_timesteps / N_between_steps;
  int N_save_atoms = 5;

  double t, E_kin, virial;

  double (*pos)[3]      = malloc(sizeof(double[N_atoms][3]));
  double (*pos_0)[3]    = malloc(sizeof(double[N_atoms][3]));//for displacements
  double (*momentum)[3] = malloc(sizeof(double[N_atoms][3]));
  double (*forces)[3]   = malloc(sizeof(double[N_atoms][3]));
  double (*displacements)[N_save_atoms] =
    malloc(sizeof(double[N_save_timesteps][N_save_atoms]));
  double (*pos_all)[N_atoms][3] =
    malloc(sizeof(double[N_save_timesteps][N_atoms][3]));
  double (*vel_all)[N_atoms][3] =
    malloc(sizeof(double[N_save_timesteps][N_atoms][3]));
  double *temperature   = malloc(sizeof(double[N_timesteps]));
  double *pressure      = malloc(sizeof(double[N_timesteps]));
  double *msd           = malloc(sizeof(double[N_save_timesteps]));
  double *vel_corr      = malloc(sizeof(double[N_save_timesteps]));
  double *pow_spec      = malloc(sizeof(double[N_save_timesteps]));
  double *freq      = malloc(sizeof(double[N_save_timesteps]));

  // Initialize to 0
  for (int i = 0; i<N_save_timesteps; i++){
    msd[i] = 0;
    pow_spec[i] = 0;
    vel_corr[i] = 0;
  }
  FILE *file_pointer;

  // read positions, momenta and cell_length
  sprintf(file_name,"../data/INIDATA_temp-%d_pres-%d.bin",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "rb");
  fread(pos, sizeof(double), 3*N_atoms, file_pointer);
  fread(momentum, sizeof(double), 3*N_atoms, file_pointer);
  fread(&cell_length, sizeof(double), 1, file_pointer);
  fclose(file_pointer);

  for (int i=0; i<N_atoms; i++){
    for (int j=0; j<3; j++){
      pos_0[i][j]=pos[i][j];
    }
  }
  inv_volume = pow(N_cells*cell_length, -3);
  get_forces_AL( forces, pos, cell_length, N_atoms); //initial cond forces

  printf("Initialized. Starting with Verlet timestepping.\n");
  for (int i=0; i<N_timesteps; i++){
    /*
       The loop over the timesteps first takes a timestep according to the
       Verlet algorithm, then calculates the energies and temeperature.
    */
    timestep_Verlet(N_atoms, pos,  momentum, forces, m_Al, dt, cell_length);

    E_kin  = get_kin_energy(N_atoms, momentum, m_Al );
    virial = get_virial_AL(pos, cell_length, N_atoms);
    /* PV = NkT + virial */
    pressure[i] = inv_volume * (1.5*E_kin + virial);
    /* 3N*kB*T/2 = 1/(2m) * \sum_{i=1}^{N} p_i^2  = p_sq/(2m) */
    temperature[i] =  E_kin * 1/(1.5*N_atoms*kB);

    if (i % N_between_steps == 0){
      int k = i/N_between_steps; // number of saved timesteps so far
      // Saves the displacements of some atoms into `displacements`
      get_displacements (N_save_atoms,  pos, pos_0, displacements[k]);

      // Saves all the positions
      copy_mat(N_atoms, 3, pos, pos_all[k]);

      // Saves all the velocities
      copy_mat(N_atoms, 3, momentum, vel_all[k]);
      //But we need to scale the momenta to get the velocities
      scale_mat(N_atoms, 3, vel_all[k], 1/m_Al);
    }

    if ((i*10) % N_timesteps == 0){ //Print out progress at every 10%
      printf("done %d0%% of Verlet timestepping\n", (i*10)/N_timesteps);
    }
  }
  printf("done 100%% of Verlet timestepping\n");

  //Calculating MSD
  printf("calculating MSD\n");
  get_MSD(N_atoms, N_save_timesteps, pos_all, msd);

  //Calculating the velocity correlation function
  printf("calculating velocity correlation\n");
  get_vel_corr(N_atoms, N_save_timesteps, vel_all, vel_corr);

  //Calculating the velocity power spectrum
  printf("calculating power spectrum\n");
  get_powerspectrum(N_atoms, N_save_timesteps, vel_all, pow_spec);
  fft_freq(freq, dt, N_save_timesteps);

  printf("writing to file\n");
  /* Write tempertaure to file */
  sprintf(file_name,"../data/temp-%d_pres-%d_Prod-test.tsv",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_timesteps; i++){
    t = i*dt; // time at step i
    fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n",
        t, temperature[i],pressure[i]);
  }
  fclose(file_pointer);

  /* Write displacements to file */
  sprintf(file_name,"../data/temp-%d_pres-%d_displacements.tsv",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "w");
  for (int i=0; i<N_save_timesteps; i++){
    t = i*dt*N_between_steps; // time at step i
    fprintf(file_pointer, "%.4f", t);
    for (int j=0; j<N_save_atoms; j++){
      fprintf(file_pointer, "\t %.8f", displacements[i][j]);
    }
    fprintf(file_pointer, "\n");
  }
  fclose(file_pointer);

  /* Write MSD to file */
  sprintf(file_name,"../data/temp-%d_pres-%d_dynamicProperties.tsv",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "w");
  // write header
  fprintf(file_pointer, "%% t[ps] \t MSD[A^2] \t vel_corr [A/ps]^2 \n");
  for (int i=0; i<N_save_timesteps; i++){
    t = i*dt*N_between_steps; // time at step i
    fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n", t, msd[i], vel_corr[i]);
  }
  fclose(file_pointer);

  /* Write power spectrum to file */
  sprintf(file_name,"../data/temp-%d_pres-%d_power-spectrum.tsv",
      (int) T_eq_C, (int) P_eq_bar);
  file_pointer = fopen(file_name, "w");
  // write header
  fprintf(file_pointer, "%% f[1/ps] \t P[A/ps]^2 \n");
  for (int i=0; i<N_save_timesteps/2; i++){ // only save from f=0 to f_crit
    fprintf(file_pointer, "%.4f \t %.8f \n", freq[i], pow_spec[i]);
  }
  fclose(file_pointer);

  // Freeing all the memory
  free(pos);           pos = NULL;
  free(pos_0);         pos_0 = NULL;
  free(momentum);      momentum = NULL;
  free(forces);        forces = NULL;
  free(temperature);   temperature = NULL;
  free(pressure);      pressure = NULL;
  free(displacements); displacements = NULL;
  free(pos_all);       pos_all = NULL;
  free(vel_all);       vel_all = NULL;
  free(msd);           msd = NULL;
  free(vel_corr);      vel_corr = NULL;
  free(pow_spec);      pow_spec = NULL;
  free(freq);          freq = NULL;

  return 0;
}
```

A.6 Misc functions : `funcs.c`

```c
#include "funcs.h"

void add_noise(int M, int N, double mat[M][N], double noise_amplitude )
{
  const  gsl_rng_type *T;    // static  info  about  rngs
  gsl_rng *q;                // rng  instance
  gsl_rng_env_setup ();      // setup  the  rngs
  T = gsl_rng_default;       // specify  default  rng
  q = gsl_rng_alloc(T);      // allocate  default  rng
  gsl_rng_set(q,time(NULL)); // Initialize  rng

  // Loops over all the elemtens in the matrix, to which we want to add noise
  for (int i=0; i<N; i++){
    for (int j=0; j<M; j++){
      // adds uniformly distributed random noise in range +-`noise_amplitude`
      mat[i][j] += noise_amplitude * (2*gsl_rng_uniform(q)-1);
    }
  }
  gsl_rng_free(q); // deallocate  rng
}

void timestep_Verlet ( int N_atoms, double (*pos)[3],  double (*momentum)[3],
               double (*forces)[3], double m, double dt,
               double cell_length){
```

```
  for (int i = 0; i < N_atoms; i++) {
    // Half-steps the momentum, then steps the position
    for (int j = 0; j < 3; j++){
      // p(t+dt/2)
      momentum[i][j] += dt * 0.5 * forces[i][j];
      // q(t+dt)
      pos[i][j] += dt * momentum[i][j] / m;
    }
  }
  // Updates the forces, based on the new positions
  // F(t+dt)
  get_forces_AL( forces, pos, cell_length, N_atoms);
  // Another half-step in the momenta
  for (int i = 0; i < N_atoms; i++) {
    for (int j = 0; j < 3; j++) {
      // p(t+dt/2)
      momentum[i][j] += dt * 0.5 * forces[i][j];
    }
  }
}

double get_kin_energy ( int N_atoms,  double (*momentum)[3], double m ) {
  double p_sq=0; // momentum squared
  for (int i = 0; i < N_atoms; i++) {
    for (int j = 0; j < 3; j++) {
      p_sq += momentum[i][j] * momentum[i][j];
    }
  }
  // E_kin = p^2/(2m)
  return p_sq / (2*m);
}

void get_displacements ( int N_atoms,  double (*positions)[3],
             double (*initial_positions)[3], double disp[]) {
  for (int i = 0; i < N_atoms; i++) {
    for (int j = 0; j < 3; j++) {
      disp[i] += (positions[i][j] - initial_positions[i][j])
    *(positions[i][j] - initial_positions[i][j]);
    }
    disp[i] = sqrt(disp[i]);
  }
}


void get_MSD ( int N_atoms,  int N_times, double all_pos[N_times][N_atoms][3],
           double MSD[N_times]) {
  // all_pos = positions of all particles at all (saved) times
  // outer time index it starts at outer it = 1, since MSD[0] = 0
  for (int it = 1; it < N_times; it++) { //
    for (int jt = 0; jt < N_times-it; jt++) { // summed time index
      for (int kn = 0; kn < N_atoms; kn++) { // particle index
          for (int kd = 0; kd < 3; kd++) {    // three dimensions
            MSD[it] += (all_pos[it+jt][kn][kd] - all_pos[jt][kn][kd])
              *(all_pos[it+jt][kn][kd] - all_pos[jt][kn][kd]);
          }
      }
    }
    MSD[it] *= 1/( (double)N_atoms * (N_times-it));
  }
}

void get_vel_corr ( int N_atoms,  int N_times,
            double all_vel[N_times][N_atoms][3],
            double vel_corr[N_times]) {
  /* all_vel = velocity of all particles at all (saved) times */
  for (int it = 0; it < N_times; it++) { //
    for (int jt = 0; jt < N_times-it; jt++) { // summed time index
      for (int kn = 0; kn < N_atoms; kn++) { // particle index
          for (int kd = 0; kd < 3; kd++) { // three dimensions
            vel_corr[it] += (all_vel[it+jt][kn][kd] * all_vel[jt][kn][kd]);
          }
      }
    }
    vel_corr[it] *= 1/( (double)N_atoms * (N_times-it));
  }
}

void get_powerspectrum ( int N_atoms, int N_times,
             double all_vel[N_times][N_atoms][3],
             double pow_spec[N_times]) {
  /* all_vel = velocity of all particles at all (saved) times */

  double vel_component[N_times]; // temp. var. "all_vel[:][i][j]"
  double pow_spec_component[N_times]; // temp. var. one component F-transf.
  double normalization_factor = 1/((double) N_atoms*N_times);

  for (int kn = 0; kn < N_atoms; kn++){// for particle index
    for (int kd = 0; kd < 3; kd++){ // for 3D
      // Copies the velocity component of one particle, into temporary variable
      for (int it = 0; it < N_times; it++){
          vel_component[it] = all_vel[it][kn][kd];
      }
      //Calculates the power spect. of this one velocity component
      powerspectrum(vel_component, pow_spec_component, N_times);
      //Adds the powerspectrum to the "output" variable
      for (int iw = 0; iw < N_times; iw++) { // for all frequencies
          pow_spec[iw] += pow_spec_component[iw];
      }// end for all frequencies
    }// end for 3D
  }// end for particle index
  for (int iw = 0; iw < N_times; iw++) { // for all frequencies
    pow_spec[iw] *= normalization_factor;
  }
}



void copy_mat (int M, int N, double mat_from[M][N], double mat_to[M][N]){
  /* Copies all matrix elements of `mat_from` to `mat_to` */
  //loops over all indices
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      mat_to[i][j] = mat_from[i][j];
    }
  }
}

void set_zero (int M, int N, double mat[M][N]){
  /* Sets all the elements of matrix `mat` to zero */
  //loops over all indices
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      mat[i][j] = 0;
    }
  }
}

void scale_mat (int M, int N, double mat[M][N], double alpha){
  /* Scales the matrix `mat` by factor `alpha` */
  //loops over all indices
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      mat[i][j] *= alpha;
    }
  }
}
```

B Auxiliary

B.1 Makefile

```

CC = gcc
CFLAGS = -O3 -Wall -Wno-unused-result

LIBS = -lm -lgsl -lgslcblas

HEADERS = initfcc.h alpotential.h funcs.h fft_func.h
OBJECTS = initfcc.o alpotential.o funcs.o fft_func.o


%.o: %.c $(HEADERS)
    $(CC) -c -o $@ $< $(CFLAGS)

all: Task1 Task2 Task3 main_Prod.c

Task1: $(OBJECTS) main_T1.c
    $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

Task2: $(OBJECTS) main_T2.c
    $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

Task3: $(OBJECTS) main_T3.c
    $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

Prod: $(OBJECTS) main_Prod.c
    $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

# $(PROGRAMS): $(OBJECTS) main_T1.c
#   $(CC) -o $@ $^ $(CFLAGS) $(LIBS)

clean:
    rm -f *.o
    touch *.c
```

C MATLAB scripts

C.1 Analysis scripts for tasks 3-7: `Al_energies.m`

```
%% initial

tmp = matlab.desktop.editor.getActive; %% cd to current path
cd(fileparts(tmp.Filename));
set(0,'DefaultFigureWindowStyle','docked');
warning('off','MATLAB:handle_graphics:exceptions:SceneNode'); % interpreter
GRAY = 0.7*[0.9 0.9 1];
AMU = 1.0364e-4;
m_Al = 27*AMU;
%% task 1: lattice energies
clc

energy_data = load('../data/lattice_energies.tsv');
a0 = energy_data(:,1);
v0 = a0.^3;

energy = energy_data(:,2);
figure(1);clf;
plot(v0,energy, 'xk');

start_v = 64;
end_v = 68;
indToInclude = (v0 > start_v) & (v0 < end_v);
p = polyfit(v0(indToInclude),energy(indToInclude),2);
hold on;

vvec = linspace(start_v, end_v);
plot(vvec, p(1)*vvec.^2 + p(2)*vvec + p(3), '-r');
xlim([64 68]);

v_min = -p(2)/(2*p(1));
a_min = v_min^(1/3);
omega_res = sqrt(2*p(1)*a_min^4/m_Al);
f_res = omega_res/(2*pi); % rough estimation of resonance frequency (?)

h1 = plot( v_min*[1 1], ax.YLim, '--k'); % plot vertical line at v_min

ax = gca; ax.YLim = [-13.45 -13.42];
ax.YTick = (-13.45:0.01:-13.42);
ylabel('$E_{\rm pot}$ [eV/unit cell]');
xlabel('$a_0^3$ [\AA$^3$]');
legend('data', 'quadratic fit', ['$V_{\rm eq} \approx \, $' ...
    num2str(round(v_min,2)) '\, \AA$^3$'], ...
    'location', 'southeast')
ax = gca; ax.Children = ax.Children(3:-1:1);
ImproveFigureCompPhys(gcf); h1.LineWidth = 2; setFigureSize(gcf, 300, 600);
saveas(gcf, '../figures/potential_energy.eps', 'epsc')
%% task 2: find a suitable timestep
clc;

dt=[1e-2,5e-3,2e-3,1e-3];
t_eq=0.5;

figure(1);clf;figure(2);clf;

for i=1:4
    T_data = load(sprintf('../data/temperature_dt-%0.0e_Task2.tsv',dt(i)));
    E_data =load(sprintf('../data/total_energy_dt-%0.0e_Task2.tsv',dt(i)));
    t = T_data(:,1);
    T = T_data(:,2);
    E = E_data(:,2);

    fprintf('dt = %0.0e\n',dt(i));

    T_avg=mean(T(t>t_eq));
    T_std=std(T(t>t_eq));
    fprintf('\tT = %0.2f +- %0.1f %%\n', T_avg, abs(T_std/T_avg)*100);

    E_avg=mean(E(t>t_eq));
    E_std=std(E(t>t_eq));
    fprintf('\tE = %0.2f +- %0.1e %%\n', E_avg, abs(E_std/E_avg)*100);

    figure(1)
    plot(t, T); hold on;

    figure(2)
    plot(t, E);hold on;
end
for ifig = 1:2
    figure(ifig);
    h = legend(strcat({'$dt = $ '}, num2str(round(dt',4)) , ' ps'));
    xlabel('$t$ [ps]');
    ax = gca;
```

```matlab
    if ifig ==1
        ylabel('$T$ [K]')
        ax.YLim = [400 1800];
    else
        ylabel('$E_{\rm tot}$ [eV/unit cell]');
        ax.YTick = (-13:0.1:-10);
        ax.YLim = [-12.6 -12.0];
    end
    ImproveFigureCompPhys(gcf,'Linewidth', 2);setFigureSize(gcf, 400, 400);
end
saveas(1, '../figures/dt-scan-temperature.eps', 'epsc')
saveas(2, '../figures/dt-scan-energy.eps', 'epsc')
%% task 3: temperature and pressure equilibration,
% and task4: test production pressure and temperature

clc; clf;
temps = [500 700 500 700];
temperatures_str = num2str([500;700]);
FILENAMES = [strcat({'../data/temp-'}, temperatures_str,...
    '_pres-1_Task3.tsv');
    strcat({'../data/temp-'}, temperatures_str, '_pres-1_Prod-test.tsv')];
bar = 6.2415e-07;
Kelvin_to_degC = -273.15;
t_eqs = [1 1 0.5 0.5]; % approximate equilibration time
N_average_points = 50;
dt = 5e-3;
tau_equilibration = 100*dt;

for iFile = 1:numel(FILENAMES)
    figure(iFile);clf;
    data = load(FILENAMES{iFile});

    t = data(:,1);
    T = data(:,2)+Kelvin_to_degC;
    P = data(:,3)/bar;

    t_eq=t_eqs(iFile);

    %fprintf('dt = %0.0e\n',dt(i));
    T_avg=mean(T(t>t_eq));
    T_std=std(T(t>t_eq));
    fprintf('\tT = %0.2f +- %0.1f K\n', T_avg, abs(T_std));

    P_avg=mean(P(t>t_eq));
    P_std=std(P(t>t_eq));
    fprintf('\tP = %0.2f +- %0.1f bar\n', P_avg, abs(P_std));

    yyaxis left

    if iFile <=2 % equlibration run, otherwise production
        plot(t./tau_equilibration,T, 'color', GRAY),hold on;
        plot(t./tau_equilibration, movmean(T,N_average_points),'-k')
    else
        plot(t,T, 'color', GRAY),hold on;
        plot(t, cumsum(T)./(1:length(t))','-k')
    end
    ylabel('$T \, [^\circ \rm C]$')

    if iFile <=2 % equlibration run, otherwise production
        ylim(temps(iFile)*(1+ 0.3*[-1,1.2]))
        yyaxis right
        plot(t./tau_equilibration,P),hold on;
        plot(t./tau_equilibration, movmean(P,N_average_points),'-k')
        legend('$\mathcal{T}$', 'mov avg','$\mathcal{P}$', 'mov avg');
        xlabel('$t/\tau_{\rm eq}$')
        %xlim([0 5])
    else
        ylim(temps(iFile)+ 100*[-3,3])
        yyaxis right
        plot(t,P),hold on;
        plot(t, cumsum(P)./(1:length(t))','-k')
        legend('$\mathcal{T}$', 'cum avg','$\mathcal{P}$', 'cum avg');
        xlabel('$t$\, [ps]')
    end
    ylabel('$P \,[\rm bar]$')
    ylim([-100,400])
    ImproveFigureCompPhys(gcf, 'linewidth', 3, 'LineColor', ...
        {'MYORANGE', GRAY, 'MYBLUE', GRAY}');
    setFigureSize(gcf, 400, 400);
end

saveas(1, '../figures/TP-eq-500.eps', 'epsc')
saveas(2, '../figures/TP-eq-700.eps', 'epsc')
saveas(3, '../figures/TP-prod-500.eps', 'epsc')
saveas(4, '../figures/TP-prod-700.eps', 'epsc')
%% determine displacements and MSD
temperatures_str = num2str([500;700]);
clc; clf;
figure(10); clf;
FILENAMES = strcat({'../data/temp-'}, temperatures_str, ...
    '_pres-1_displacements.tsv');
```

```
FILENAMES_Dyn = strcat({'../data/temp-'}, temperatures_str, ...
    '_pres-1_dynamicProperties.tsv');
FILENAMES_Pow = strcat({'../data/temp-'}, temperatures_str, ...
    '_pres-1_power-spectrum.tsv');
for iFile = 1:numel(FILENAMES)
    figure(iFile); clf;
    data = load(FILENAMES{iFile});
    t = data(:,1);
    dx = data(:,2:end);

    data = load(FILENAMES_Dyn{iFile});
    MSD = data(:,2);
    vel_corr = data(:,3);
    plot(t, MSD, 'k'); hold on;

    if iFile ==2 % liquid
        tStart = 1;
        D = MSD(t>tStart)./(6*t(t>tStart));
        selfDiffusionCoeff = mean(D); % in   ^2 /ps
        plot(t, 6*t*selfDiffusionCoeff, ':r');
    end

    plot(t, dx.^2, 'color', GRAY); hold on;

    xlabel('$t$ [ps]')
    ylabel('$\Delta x^2 \,[\rm \AA^2]$')
    if iFile ==1
        ylim([ 0 1.0]);
        leg = legend( '$\Delta_{\rm MSD}$', 'individual trajectories');
    else
        ylim([0 20]);
        leg = legend('$\Delta_{\rm MSD}$', '$6 t D_s$', ...
            'individual trajectories');
    end

    leg.Location='northwest';
    ImproveFigureCompPhys(gcf, 'Linewidth', 2);
    ax = gca; [ax.Children(6:end).LineWidth] = deal(5);
    ax.Children = ax.Children([6:end 1:5]);
    setFigureSize(gcf, 400, 400);
end

% velocity correlation
figure(10);clf; figure(11);clf;
n_average_points = 1;%30;
for iFile = 1:numel(FILENAMES)
    data = load(FILENAMES_Dyn{iFile});
    t = data(:,1);
    vel_corr = data(:,3);

    data = load(FILENAMES_Pow{iFile});
    freq = data(:,1);
    pow_spec = data(:,2);

    figure(10);
    plot(t, vel_corr/vel_corr(1)); hold on;

    dt = t(2)-t(1);
    N_times = round(length(t)/2);% we have bad statistics at later times.
    deltaf = 1/(N_times * dt);
    freqvec = 0:deltaf:(1/(2*dt));
    PhiHat = 2 * trapz(t(1:N_times), ...
        (vel_corr(1:N_times) * ones(size(freqvec))) .* ...
        cos(2*pi*t(1:N_times) * freqvec ), 1);

    figure(11);
    plot(freqvec, m_Al/2*PhiHat); hold on;
    plot(freq, m_Al/2* pow_spec*t(end), ':'); hold on;
    if iFile ==2 % liquid
        tStart = 1;
        selfDiffusionCoeff_spectral = PhiHat(1)/6; % in   ^2 /ps
    end

end

disp([selfDiffusionCoeff selfDiffusionCoeff_spectral]);

figure(10)
xlim([0 1]);
leg = legend(strcat({'$T='}, num2str([500;700]), '\,^\circ $C'));
leg.Location='northeast';
xlabel('$t$ [ps]')
ylabel('$\Phi (t)/\Phi(0)$')
ImproveFigureCompPhys(gcf,'LineColor', {'MYRED', 'MYLIGHTBLUE'}');
setFigureSize(gcf, 400, 400);

figure(11)
leg = legend('$T= 500 \, ^\circ $C, $ \hat \Phi$' ,...
    '$T= 500 \, ^\circ $C, $\hat P$',...
    '$T= 700 \, ^\circ $C, $ \hat \Phi$', ...
    '$T= 700 \, ^\circ $C, $\hat P$');
```

```
xlim([0 30])
ylim([0 Inf])
xlabel('$f$ [THz]')
ylabel('$\frac{1}{2} m \hat P$ [eV/THz]')
setFigureSize(gcf, 400, 400);

ImproveFigureCompPhys(gcf,'LineColor', ...
    {'r', 'MYRED', 'GERIBLUE','MYLIGHTBLUE'}');
saveas(1, '../figures/MSD-500.eps', 'epsc')
saveas(2, '../figures/MSD-700.eps', 'epsc')
saveas(10, '../figures/Phi-t.eps', 'epsc')
saveas(11, '../figures/P-freq.eps', 'epsc')
```

C.2 Improve figure appearance: `ImproveFigureCompPhys.m`

```
function ImproveFigureCompPhys(varargin)
%ImproveFigureCompPhys Improves the figures of supplied handles
%  Input:
% - none (improve all figures) or handles to figures to improve
% - optional:
%       LineWidth  int
%       LineStyle  column vector cell, e.g. {'-','--'}',
%       LineColor  column vector cell, e.g. {'k',[0 1 1], 'MYBLUE'}'
%                            colors: MYBLUE,MYORANGE,MYGREEN,MYPURPLE, MYYELLOW,
%                            MYLIGHTBLUE, MYRED
%       Marker column vector cell, e.g. {'.', 'o', 'x'}'

% ImproveFigure was originally written by Adam Stahl, but has been heavily
% modified by Linnea Hesslow


%%% Handle inputs
% If no inputs or if the first argument is a string (a property rather than
% a handle), use all open figures
if nargin == 0 || ischar(varargin{1})
    %Get all open figures
    figHs = findobj('Type','figure');
    nFigs = length(figHs);
else
    % Check the supplied figure handles
    figHs = varargin{1};
    figHs = figHs(ishandle(figHs) == 1); %Keep only those handles that are ↩
         proper graphics handles
    nFigs = length(figHs);
end

% Define desired properties
titleSize = 24;
interpreter = 'latex';
lineWidth = 4;
axesWidth = 1.5;
labelSize = 22;
textSize = 20;
legTextSize = 18;
tickLabelSize = 18;
LineColor = {};
LineStyle = {};
Marker = {};

% define colors
co =  [ 0    0.4470    0.7410
    0.8500    0.3250    0.0980
    0.9290    0.6940    0.1250
    0.4940    0.1840    0.5560
    0.4660    0.6740    0.1880
    0.3010    0.7450    0.9330
    0.6350    0.0780    0.1840 ];
colors = struct('MYBLUE', co(1,:),...
    'MYORANGE', co(2,:),...
    'MYYELLOW', co(3,:),...
    'MYPURPLE', co(4,:),...
    'MYGREEN', co(5,:),...
    'MYLIGHTBLUE', co(6,:),...
    'MYRED',co(7,:),...
    'GERIBLUE', [0.3000    0.1500    0.7500],...
    'GERIRED', [1.0000    0.2500    0.1500],...
    'GERIYELLOW', [0.9000    0.7500    0.1000],...
    'LIGHTGREEN', [0.4    0.85    0.4],...
    'LINNEAGREEN', [7 184 4]/255);

% Loop through the supplied arguments and check for properties to set.
for i = 1:nargin
    if ischar(varargin{i})
        switch lower(varargin{i})   %Compare lower case strings
            case 'linewidth'
                lineWidth = varargin{i+1};
            case 'linestyle'
                LineStyle = varargin{i+1};
            case 'linecolor'
                LineColor = varargin{i+1};
                for iLineColor = 1:numel(LineColor)
                    if isfield(colors, LineColor{iLineColor})
                        LineColor{iLineColor} = colors.(LineColor{iLineColor});
                    end
                end
            case 'marker'
                Marker = varargin{i+1};
        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% Improve the figure(s)

for iFig = 1:nFigs

    fig = figHs(iFig);

    lineObjects = findall(fig, 'Type', 'line');
    textObjects = findall(fig, 'Type', 'text');
    axesObjects = findall(fig, 'Type', 'axes');
    legObjects =  findall(fig, 'Type', 'legend');
    contourObjects = findall(fig,'Type','contour'); % not counted as lines

    %%% TEXT APPEARANCE: first set all to textSize and then change the ones
    %%% that need to be changed again

    %Change size of any text objects in the plot
    set(textObjects,'FontSize',textSize);
    set(legObjects,'FontSize',legTextSize);

    %%% FIX LINESTYLE, COLOR ETC. FOR EACH PLOT SEPARATELY
    for iAx =  1:numel(axesObjects)
        lineObjInAx = findall(axesObjects(iAx), 'Type', 'line');

        %set line style and color style (only works if all figs have some
        %number of line plots..)
        if ~isempty(LineStyle)
            set(lineObjInAx, {'LineStyle'}, LineStyle)
            set(contourObjects, {'LineStyle'}, LineStyle); %%%%%%
        end
        if ~isempty(LineColor)
            set(lineObjInAx, {'Color'}, LineColor)
            set(contourObjects, {'LineColor'}, LineColor); %%%%%%
        end
        if ~isempty(Marker)
            set(lineObjInAx, {'Marker'}, Marker)
            set(lineObjInAx, {'Markersize'}, num2cell(10+22*strcmp(Marker, '.'))↩
            )
        end

        %%% change font sizes.
        % Tick label size
        xLim = axesObjects(iAx).XLim;
        axesObjects(iAx).FontSize = tickLabelSize;
        axesObjects(iAx).XLim = xLim;
        %Change label size
        axesObjects(iAx).XLabel.FontSize = labelSize;
        axesObjects(iAx).YLabel.FontSize = labelSize;

        %Change title size
        axesObjects(iAx).Title.FontSize = titleSize;
    end

    %%% LINE APPEARANCE
    %Change line thicknesses
    set(lineObjects,'LineWidth',lineWidth);
    set(contourObjects, 'LineWidth', lineWidth);
    set(axesObjects, 'LineWidth',axesWidth)

    % set interpreter: latex or tex
    set(textObjects, 'interpreter', interpreter)
    set(legObjects, 'Interpreter', interpreter)
    set(axesObjects,'TickLabelInterpreter', interpreter);
end
end
```

C.3 Change size of figures: `setFigureSize.m`

```
function [ fig ] = setFigureSize( fig, H, W )
fig.Units = 'points';
fig.WindowStyle = 'normal'; % undock
fig.Position(3:4) = [W H];
end
```