

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 23, 2018

Task N ^o	Points	Avail. points
Σ		

Introduction

The velocity Verlet algorithm is a semi-implicit and efficient method to simulate an 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 an fcc lattice. By scaling the positions, momenta and lattice parameter, we can equilibrate the temperature and pressure to prescribed values. We study the aluminum system at 500 °C and 700 °C, which correspond to the solid and liquid state respectively, and compute two dynamic quantities: mean squared displacements and the velocity correlation function.

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 energy¹, and obtained $V_{eq} \approx 65.38 \text{ \AA}^3$. This corresponds to the equilibrium lattice parameter $a_{eq} \approx 4.029 \text{ \AA}$ at 0 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.

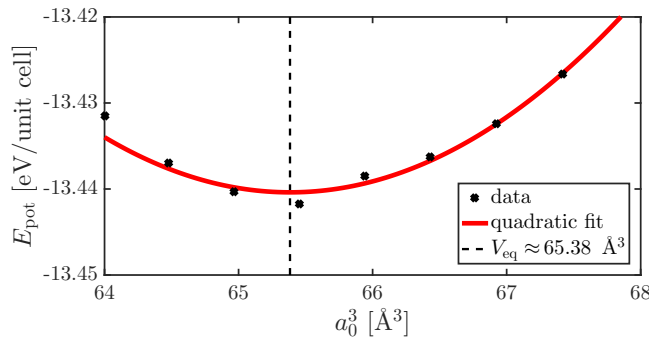


Figure 1: The potential energy per unit cell for aluminum as a function of the lattice parameter cubed.

Task 2: determine the time step

With the random noise, the temperature and the energy differ between runs, but are in the same order of magnitude. From figure 2, we determine that $dt = 5 \cdot 10^{-3} \text{ ps} = 5 \text{ fs}$ is a sufficient time step. This is in line with the lecture notes, where it is stated that a suitable timestep would normally be a few femtoseconds, or somewhat larger for heavy atoms. 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.

Tasks 3 and 4: temperature and pressure equilibration

We set $\tau_P = \tau_T = 100dt$, where $dt = 5 \cdot 10^{-3} \text{ ps}$, 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 the system was first melted by increasing the temperature to 900 °C. To

¹We 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 $a \approx a_0$.

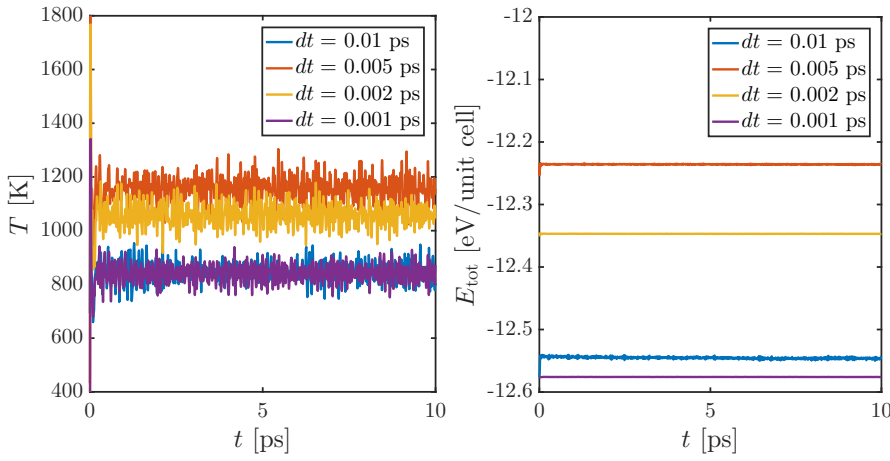


Figure 2: The temperature and kinetic energy per unit cell as a function of time for four different timesteps.

Table 1: 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}	$847 \pm 4.2\%$	$-12.55 \pm 1.2 \cdot 10^{-2}\%$
$5 \cdot 10^{-3}$	$1157 \pm 3.8\%$	$-12.24 \pm 3.6 \cdot 10^{-3}\%$
$2 \cdot 10^{-3}$	$1058 \pm 3.7\%$	$-12.35 \pm 6.6 \cdot 10^{-4}\%$
$1 \cdot 10^{-3}$	$841 \pm 3.7\%$	$-12.58 \pm 3.6 \cdot 10^{-4}\%$

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

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

where $1 \text{ bar} = 6.2415 \cdot 10^{-7} \text{ eV/\AA}^3$ in atomic units. However, we set $\kappa = 100\kappa_{\text{Al}}$ since the pressure equilibration happened on a much longer timescale than τ_P with $\kappa = \kappa_{\text{Al}}$. We have not yet figured out why this is.

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

$$a_0 \approx 4.10 \text{ \AA}, \quad T = 500^\circ\text{C}, \quad (2)$$

$$a_0 \approx 4.29 \text{ \AA}, \quad T = 700^\circ\text{C}. \quad (3)$$

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.

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 timestep to $dt = 5 \cdot 10^{-4} \text{ ps}$ and the simulation length to $t_{\text{end}} = 5 \text{ ps}$ 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.

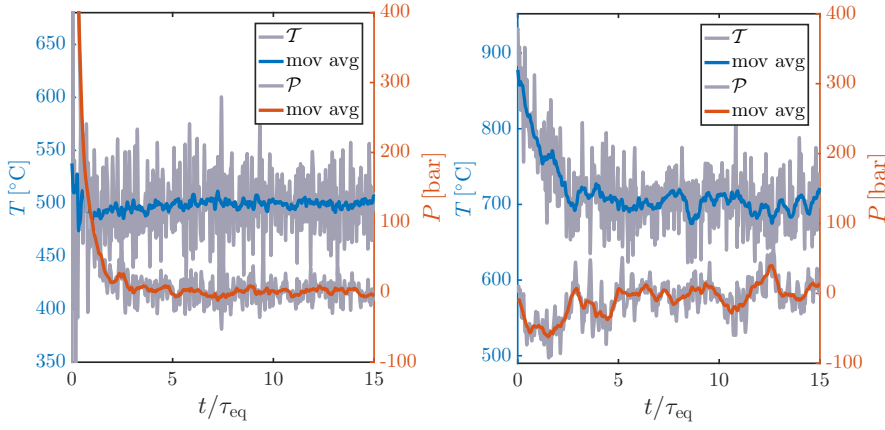


Figure 3: The instantaneous values of \mathcal{T} and \mathcal{P} overlaid with with a moving average using 100 time steps, which corresponds to $\Delta t = \tau_P/2$. Left panel: $T = 500^\circ\text{C}$, right panel: $T = 700^\circ\text{C}$.

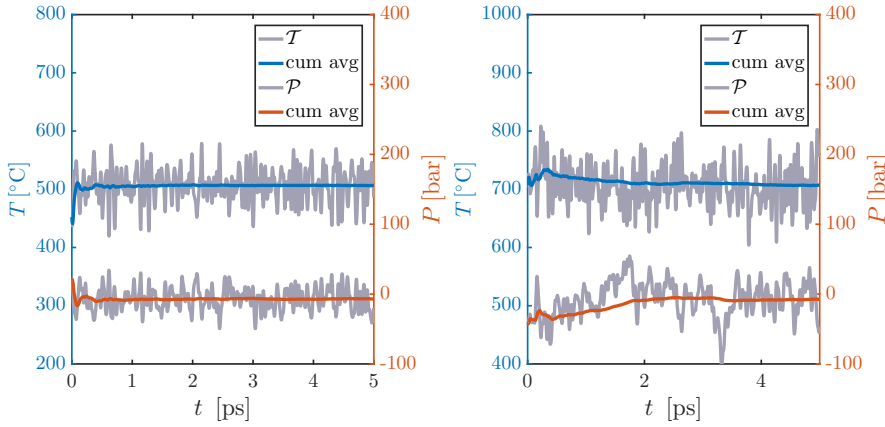


Figure 4: The instantaneous values, and the cumulative averages, of the temperature and the pressure in the production runs. Left panel: $T = 500^\circ\text{C}$, right panel: $T = 700^\circ\text{C}$

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

$$\Delta_{\text{MSD}}(t) = \lim_{T \rightarrow \infty} \frac{1}{T} \int_0^T dt' \frac{1}{N_{\text{atoms}}} \sum_{i=0}^{N_{\text{atoms}}-1} [\mathbf{r}_i(t+t') - \mathbf{r}_i(t')]^2 \quad (4)$$

\Rightarrow

$$\Delta_{\text{MSD}}(t_k) \approx \frac{1}{N_T - k} \frac{1}{N_{\text{atoms}}} \sum_{j=0}^{N_T-k-1} \sum_{i=0}^{N_{\text{atoms}}-1} [\mathbf{r}_i(t_{k+j}) - \mathbf{r}_i(t_j)]^2 \quad (5)$$

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 (5). 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 ($\Delta_{\text{MSD}} \propto \sqrt{t}$). 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 $D_s \approx 0.52 \text{ \AA}^2/\text{ps}$.

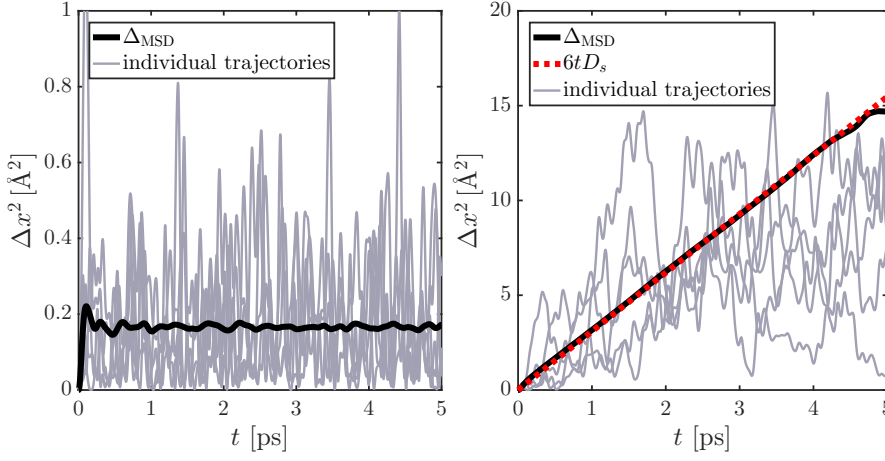


Figure 5: Five individual particle trajectories (gray thin lines), overlaid with the mean squared displacement (thick black line). In the left panel, $T = 500\text{ }^{\circ}\text{C}$, the system is in a solid state. In the right panel, $T = 700\text{ }^{\circ}\text{C}$, the system is in a liquid state,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

$$\Phi_j = \frac{1}{N-j} \sum_{i=0}^{N-j-1} \langle v_{i+j} v_i \rangle, \quad (6)$$

where $j = 0, 1, \dots, N-1$ 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.

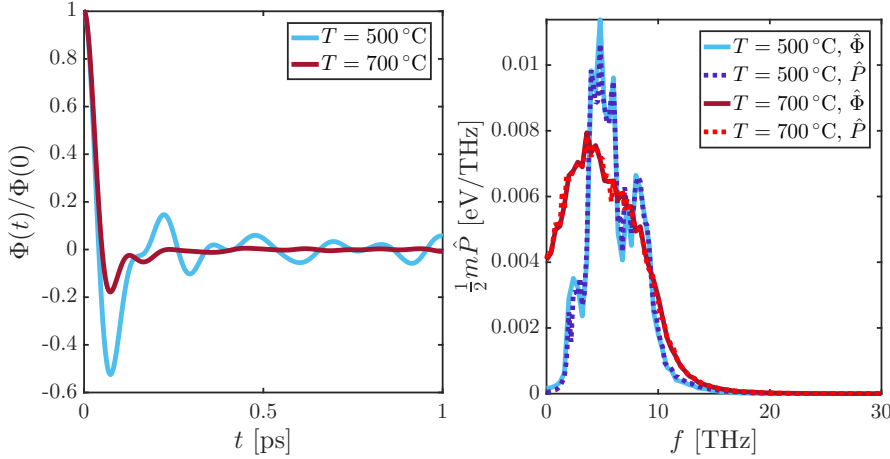


Figure 6: Left panel: The velocity correlation function, and (right panel) its spectrum, calculated both directly from the velocity correlation (solid line) and from the power spectrum of the particle velocity (dotted line). Blue lines show $T = 500\text{ }^{\circ}\text{C}$ and red lines $T = 700\text{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

We then proceeded to numerically approximate the integral

$$\hat{\Phi}(f) = 2 \int_0^{\infty} dt \Phi(t) \cos(2\pi f t) \approx 2 \int_0^{T_s} dt \Phi(t) \cos(2\pi f t) \quad (7)$$

using a trapezoidal method in MATLAB, with a frequency range $f = 0$ to $f = 1/(2\Delta t) = f_{\text{Nyquist}}$, and frequency steps $\Delta f = 1/T_s$, 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

$$\begin{aligned}\hat{P}(\omega) &= \lim_{T \rightarrow \infty} \frac{1}{T} \left\langle \left| \int_0^T dt v(t) e^{i\omega t} \right|^2 \right\rangle \\ &\approx \frac{1}{T} \left\langle \left| \int_0^T dt v(t) e^{i\omega t} \right|^2 \right\rangle \\ \Rightarrow \hat{P}_k &= \frac{1}{T} \left\langle \left| \frac{T}{N} \sum_{i=0}^{N-1} v_i \exp\left(i2\pi \frac{ik}{N}\right) \right|^2 \right\rangle = \frac{T}{N} \langle |\hat{v}_k|^2 \rangle\end{aligned}\tag{8}$$

where the averages is taken over all atoms, and

$$\hat{v}_k = \sqrt{N} \sum_{i=0}^{N-1} v_i \exp\left(i2\pi \frac{ik}{N}\right)\tag{9}$$

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-Khinchine 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 (6). Therefore, even though the results are similar, the power spectrum method in equation (8), 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 $D_s = 0.49 \text{ \AA}^2/\text{ps}$, 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

```
1  /*
2  main_T1.c Task 1 H1b
3  In this task, we scan over a range of lattice parameters, a0, to determine
4  which one results in the lowest potential energy stored in the lattice.
5
6  System of units:
7  Energy   - eV
8  Time     - ps
9  Length   - Angstrom
10 Temp     - K
11 Mass     - eV (ps)^2 A^(-2)
12 Pressure - eV A^(-3)
13 */
14 #include <stdio.h>
15 #include <math.h>
16 #include <stdlib.h>
17
18 #include "initfcc.h"
19 #include "alpotential.h"
20
21 #define N_cells 4
22 #define N_lattice_params 25
23
24 /* Main program */
25 int main()
26 {
27     int N_atoms = 4*N_cells*N_cells*N_cells;
28     double a0;
29     double a0_min = 4.0;
30     double a0_max = 4.2;
31     double da0 = (a0_max - a0_min)/N_lattice_params;
32
33     double (*pos)[3] = malloc(sizeof(double[N_atoms][3]));
34     double *energy = malloc(sizeof(double[N_lattice_params]));
35
36     FILE *file_pointer;
37
38     for (int i=0; i<N_lattice_params; i++){
39         a0 = a0_min + i*da0; // The lattice constant of this iteration
40         init_fcc(pos, N_cells, a0); // Init, FCC cells with lattice constant `a0`
41         // energy per unit cell
42         energy[i] = get_energy_AL(pos, N_cells*a0, N_atoms )*4/N_atoms;
43     }
44
45     // Write to files
46     file_pointer = fopen("../data/lattice_energies.tsv", "w");
47     for (int i=0; i<N_lattice_params; i++){
48         a0 = a0_min + i*da0;
49         fprintf(file_pointer, "%.8f \t %.8f \n", a0, energy[i]);
50     }
51     fclose(file_pointer);
52
53     free(pos); pos = NULL;
54     free(energy); energy = NULL;
55     return 0;
56 }
```

A.2 Main program Task 2: main.T2.c

```
1  /*
2  main_T2.c, Task 2, H1b
3  In this task, we add random noise to the particle positions and see how the
4  system evolves in time. Using the kinetic energy of the particles, we can
5  derive an instantaneous temperature of the system.
6
7  System of units:
8  Energy   - eV
9  Time     - ps
10 Length   - Angstrom
11 Temp     - K
12 Mass     - eV (ps)^2 A^(-2)
13 Pressure - eV A^(-3)
14 */
15
16 #include <stdio.h>
17 #include <math.h>
18 #include <stdlib.h>
19 #include <time.h>
20
```

```

21 #include "initfcc.h"
22 #include "alpotential.h"
23 #include "funcs.h"
24
25 #define N_cells 4
26 #define AMU 1.0364e-4
27 #define kB 8.6173303e-5
28
29 /* Main program */
30 int main()
31 {
32     int N_atoms = 4*N_cells*N_cells*N_cells;
33     double m_Al = 27*AMU;
34
35     double a_eq = 4.03; // Min potential energy lattice constant
36
37     double noise_amplitude = 6.5e-2 * a_eq;
38     double t_max=10; //
39     double dt = 1e-3;
40     int N_timesteps = t_max/dt;
41     double t, E_kin;
42
43     double (*pos)[3] = malloc(sizeof(double[N_atoms][3]));
44     double (*momentum)[3] = malloc(sizeof(double[N_atoms][3]));
45     double (*forces)[3] = malloc(sizeof(double[N_atoms][3]));
46     double *temperature = malloc(sizeof(double[N_timesteps]));
47     double *E_tot = malloc(sizeof(double[N_timesteps]));
48
49     FILE *file_pointer;
50
51     init_fcc(pos, N_cells, a_eq); // initialize fcc lattice
52     add_noise( N_atoms, 3, pos, noise_amplitude ); // adds random noise to pos
53     set_zero( N_atoms, 3, momentum); // set momentum to 0
54     get_forces_AL( forces, pos, a_eq*N_cells, N_atoms); //initial cond forces
55
56     for (int i=0; i<N_timesteps; i++){
57         /*
58          * The loop over the timesteps first takes a timestep according to the
59          * Verlet algorithm, then calculates the energies and temeperature.
60          */
61         timestep_Verlet (N_atoms, pos, momentum, forces, m_Al, dt, a_eq*N_cells);
62
63         E_kin = get_kin_energy(N_atoms, momentum, m_Al );
64         E_tot[i] = (E_kin + get_energy_AL(pos, a_eq*N_cells, N_atoms))*4/N_atoms;
65
66         /*  $3N \cdot k_B \cdot T/2 = 1/(2m) \cdot \sum_{i=1}^N p_i^2 = p_{sq}/(2m)$  */
67         temperature[i] = E_kin * 2/(3*N_atoms*kB);
68     }
69
70     /* Write tempertaure to file */
71     char file_name[100];
72     sprintf(file_name, "../data/temperature_dt-%0.0e_Task2.tsv", dt);
73     file_pointer = fopen(file_name, "w");
74     for (int i=0; i<N_timesteps; i++){
75         t = i*dt; // time at step i
76         fprintf(file_pointer, "%.4f \t %.8f \n", t, temperature[i]);
77     }
78     fclose(file_pointer);
79
80     /* Write total energy to file */
81     sprintf(file_name, "../data/total_energy_dt-%0.0e_Task2.tsv", dt);
82     file_pointer = fopen(file_name, "w");
83     for (int i=0; i<N_timesteps; i++){
84         t = i*dt; // time at step i
85         fprintf(file_pointer, "%.4f \t %.8f \n", t, E_tot[i]);
86     }
87     fclose(file_pointer);
88
89     free(pos); pos = NULL;
90     free(momentum); momentum = NULL;
91     free(forces); forces = NULL;
92     free(temperature); temperature = NULL;
93     free(E_tot); E_tot = NULL;
94     return 0;
95 }

```

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

```

1 /*
2  * main_T3.c, Tasks 3 and 4, Hib. Also used as input in Tasks 5-7.
3  * In this task, we use an equilibration scheme, based on scaling particle momenta
4  * and positions, to equilibrate the temperature and pressure in the system. We do
5  * this for T=500 degC and T=700 degC and P=1 bar. The difference between the two
6  * temperatures are that the higher temperature results in a melted system. (To

```



```

7  ensure that the system is melted properly, we first raise the temperature to
8  900 degC and then lower it back to 700 degC.)
9
10 After the system has equilibrated, we save the full phase space (all particle
11 positions and momenta) as well as the equilibrated lattice parameter to a
12 binary file which then can be read in for a production run.
13
14 System of units:
15 Energy      - eV
16 Time        - ps
17 Length      - Angstrom
18 Temp        - K
19 Mass        - eV (ps)^2 A^(-2)
20 Pressure    - eV A^(-3)
21 */
22
23 #include <stdio.h>
24 #include <math.h>
25 #include <stdlib.h>
26 #include <time.h>
27
28 #include "initfcc.h"
29 #include "alpotential.h"
30 #include "funcs.h"
31
32 #define N_cells 4
33 /* define constants in atomic units: eV, , ps, K */
34 #define AMU 1.0364e-4
35 #define degC_to_K 273.15
36 #define bar 6.2415e-07
37 #define kB 8.61733e-5
38
39 /* Main program */
40 int main()
41 {
42     char file_name[100];
43
44     int N_atoms = 4*N_cells*N_cells*N_cells;
45     double m_Al = 27*AMU;
46     /*
47      Values of Young's and shear modulus, Y and G resp., taken from
48      Physics Handbook, table T 1.1. Bulk modulus then calculated as
49      B = Y*G / (9*G - 3*Y) [F 1.15, Physics Handbook]
50      kappa = 1/B
51     */
52     double kappa_Al = 100/(6.6444e+05 * bar); // STRANGE FACTOR 100 OFF !!!
53     double a_eq = 4.03;
54     double cell_length = a_eq*N_cells;
55     double inv_volume = pow(N_cells*cell_length, -3);
56     double noise_amplitude = 6.5e-2 * a_eq;
57
58     double T_final_C = 500;
59     int nRuns = 1; //2 if melt, 1 otherwise
60     double T_melt_C = 900;
61
62     double P_final_bar = 1;
63
64     double T_eq;
65     double P_eq = P_final_bar*bar;
66     double dt = 5e-3;
67     double tau_T = 100*dt;
68     double tau_P = 100*dt;
69     double t_eq = 15*tau_P; //equilibration times
70     int N_timesteps = t_eq/dt;
71
72     double alpha_T, alpha_P, alpha_P_cube_root;
73     double t, E_kin, virial;
74
75     double (*pos)[3] = malloc(sizeof(double)[N_atoms][3]);
76     double (*momentum)[3] = malloc(sizeof(double)[N_atoms][3]);
77     double (*forces)[3] = malloc(sizeof(double)[N_atoms][3]);
78     double *temperature = malloc(sizeof(double)[N_timesteps]);
79     double *pressure = malloc(sizeof(double)[N_timesteps]);
80
81     FILE *file_pointer;
82
83
84     init_fcc(pos, N_cells, a_eq); // initialize fcc lattice
85     add_noise( N_atoms, 3, pos, noise_amplitude ); // adds random noise to pos
86     set_zero( N_atoms, 3, momentum); // set momentum to 0
87     get_forces_AL( forces, pos, cell_length, N_atoms); //initial cond forces
88
89     for (int irun=0; irun < nRuns; irun++){// last run: final, irun = 0
90         if (irun == nRuns - 1){ // final run
91             T_eq = T_final_C + degC_to_K;
92         }else{
93             T_eq = T_melt_C + degC_to_K;
94         }
95         for (int i=0; i<N_timesteps; i++){
96             /*
97              The loop over the timesteps first takes a timestep according to the

```

```

98     Verlet algorithm, then calculates the energies and temeperature.
99     */
100     timestep_Verlet(N_atoms, pos, momentum, forces, m_Al, dt, cell_length);
101
102     E_kin = get_kin_energy(N_atoms, momentum, m_Al );
103     virial = get_virial_AL(pos, cell_length, N_atoms);
104
105     /*  $3N \cdot k_B \cdot T/2 = 1/(2m) \cdot \sum_{i=1}^N p_i^2 = p_{sq}/(2m)$  */
106     temperature[i] = E_kin * 1/(1.5*N_atoms*kB);
107     /*  $PV = NkT + \text{virial}$  */
108     pressure[i] = inv_volume * (1.5*E_kin + virial);
109
110     /* Equilibrate temperature by scaling momentum by a factor sqrt(alpha_T).
111        N.B. It is equally valid to scale the momentum instead of the velocity↔
112        since they only differ by a constant factor m.
113     */
114     alpha_T = 1 + 2*dt*(T_eq - temperature[i]) / (tau_T * temperature[i]);
115     scale_mat(N_atoms, 3, momentum, sqrt(alpha_T));
116
117     // Equilibrate pressure by scaling the posistions by a factor of
118     //  $\alpha_P^{1/3}$ 
119     alpha_P = 1 - kappa_Al * dt*(P_eq - pressure[i])/tau_P;
120     alpha_P_cube_root = pow(alpha_P, 1.0/3.0);
121     scale_mat(N_atoms, 3, pos, alpha_P_cube_root);
122
123     cell_length*=alpha_P_cube_root;
124     inv_volume*=1/alpha_P;
125
126     temperature[i]*=alpha_T;
127     pressure[i]*=alpha_P;
128 }
129 }
130
131 printf("equilibrium a0 = %.4f A\n", cell_length/N_cells);
132
133 /* Write tempertaure to file */
134 sprintf(file_name, "../data/temp-%d_pres-%d_Task3.tsv",
135         (int) T_final_C, (int) P_final_bar);
136 file_pointer = fopen(file_name, "w");
137 for (int i=0; i<N_timesteps; i++){
138     t = i*dt; // time at step i
139     fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n",
140             t, temperature[i], pressure[i]);
141 }
142 fclose(file_pointer);
143
144 /* Write phase space coordinates to file */
145 sprintf(file_name, "../data/phase-space-temp-%d_pres-%d.tsv",
146         (int) T_final_C, (int) P_final_bar);
147 file_pointer = fopen(file_name, "w");
148 for (int i=0; i<N_atoms; i++){
149     for (int j=0; j<3; j++){
150         fprintf(file_pointer, " %.16e \t", pos[i][j]);
151     }
152     for (int j=0; j<3; j++){
153         fprintf(file_pointer, " %.16e \t", momentum[i][j]);
154     }
155     fprintf(file_pointer, "\n");
156 }
157 fclose(file_pointer);
158
159 /* save equilibrated position and momentum as a binary file */
160 sprintf(file_name, "../data/INIDATA-temp-%d_pres-%d.bin",
161         (int) T_final_C, (int) P_final_bar);
162 file_pointer = fopen(file_name, "wb");
163 fwrite(pos, sizeof(double), 3*N_atoms, file_pointer);
164 fwrite(momentum, sizeof(double), 3*N_atoms, file_pointer);
165 fwrite(&cell_length, sizeof(double), 1, file_pointer);
166 fclose(file_pointer);
167
168 free(pos); pos = NULL;
169 free(momentum); momentum = NULL;
170 free(forces); forces = NULL;
171 free(temperature); temperature = NULL;
172 free(pressure); pressure = NULL;
173 return 0;
174 }

```

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

```

1  /*
2  main_Prod.c, Production runs, H1b
3  In this program, we use the equilibrated micro-states from Tasks 3-4 to study
4  dynamical properties, such as mean squared displacement (MSD), velocity
5  auto-correlation function, and the power spectral density of the atom

```

```

6   movements.
7
8   System of units:
9   Energy   - eV
10  Time      - ps
11  Length    - Angstrom
12  Temp      - K
13  Mass      - eV (ps)^2 A^(-2)
14  Pressure  - eV A^(-3)
15  */
16
17  #include <stdio.h>
18  #include <math.h>
19  #include <stdlib.h>
20  #include <time.h>
21
22  #include "initfcc.h"
23  #include "alpotential.h"
24  #include "funcs.h"
25
26  #define N_cells 4
27  /* define constants in atomic units: eV, , ps, K */
28  #define AMU 1.0364e-4
29  #define degC_to_K 273.15
30  #define bar 6.2415e-07
31  #define kB 8.61733e-5
32
33  /* Main program */
34  int main()
35  {
36      char file_name[100];
37
38      int N_atoms = 4*N_cells*N_cells*N_cells;
39      double m_Al = 27*AMU;
40      double cell_length;
41      double inv_volume;
42
43      double T_eq_C = 500;
44      double P_eq_bar = 1;
45
46      double dt = 5e-4; // higher res for spectral function
47      double t_end = 5;
48      int N_timesteps = t_end/dt;
49      int N_between_steps = 1; // save all steps for max res in spectral function
50      int N_save_timesteps = N_timesteps / N_between_steps;
51      int N_save_atoms = 5;
52
53      double t, E_kin, virial;
54
55      double (*pos)[3] = malloc(sizeof(double[N_atoms][3]));
56      double (*pos_0)[3] = malloc(sizeof(double[N_atoms][3])); //for displacements
57      double (*momentum)[3] = malloc(sizeof(double[N_atoms][3]));
58      double (*forces)[3] = malloc(sizeof(double[N_atoms][3]));
59      double (*displacements)[N_save_atoms] =
60          malloc(sizeof(double[N_save_timesteps][N_save_atoms]));
61      double (*pos_all)[N_atoms][3] =
62          malloc(sizeof(double[N_save_timesteps][N_atoms][3]));
63      double (*vel_all)[N_atoms][3] =
64          malloc(sizeof(double[N_save_timesteps][N_atoms][3]));
65      double *temperature = malloc(sizeof(double[N_timesteps]));
66      double *pressure = malloc(sizeof(double[N_timesteps]));
67      double *msd = malloc(sizeof(double[N_save_timesteps]));
68      double *vel_corr = malloc(sizeof(double[N_save_timesteps]));
69      double *pow_spec = malloc(sizeof(double[N_save_timesteps]));
70      double *freq = malloc(sizeof(double[N_save_timesteps]));
71
72      // Initialize to 0
73      for (int i = 0; i < N_save_timesteps; i++){
74          msd[i] = 0;
75          pow_spec[i] = 0;
76          vel_corr[i] = 0;
77      }
78      FILE *file_pointer;
79
80      // read positions, momenta and cell_length
81      sprintf(file_name, "../data/INIDATA-temp-%d-pres-%d.bin",
82              (int) T_eq_C, (int) P_eq_bar);
83      file_pointer = fopen(file_name, "rb");
84      fread(pos, sizeof(double), 3*N_atoms, file_pointer);
85      fread(momentum, sizeof(double), 3*N_atoms, file_pointer);
86      fread(&cell_length, sizeof(double), 1, file_pointer);
87      fclose(file_pointer);
88
89      for (int i=0; i<N_atoms; i++){
90          for (int j=0; j<3; j++){
91              pos_0[i][j]=pos[i][j];
92          }
93      }
94      inv_volume = pow(N_cells*cell_length, -3);
95      get_forces_AL( forces, pos, cell_length, N_atoms); //initial cond forces
96

```

```

97 printf("Initialized. Starting with Verlet timestepping.\n");
98 for (int i=0; i<N_timesteps; i++){
99     /*
100     The loop over the timesteps first takes a timestep according to the
101     Verlet algorithm, then calculates the energies and temeperature.
102     */
103     timestep_Verlet(N_atoms, pos, momentum, forces, m_Al, dt, cell_length);
104
105     E_kin = get_kin_energy(N_atoms, momentum, m_Al);
106     virial = get_virial_AL(pos, cell_length, N_atoms);
107     /* PV = NkT + virial */
108     pressure[i] = inv_volume * (1.5*E_kin + virial);
109     /* 3N*kB*T/2 = 1/(2m) * \sum_{i=1}^N p_i^2 = p_sq/(2m) */
110     temperature[i] = E_kin * 1/(1.5*N_atoms*kB);
111
112     if (i % N_between_steps == 0){
113         int k = i/N_between_steps; // number of saved timesteps so far
114         // Saves the displacements of some atoms into `displacements`
115         get_displacements(N_save_atoms, pos, pos_0, displacements[k]);
116
117         // Saves all the positions
118         copy_mat(N_atoms, 3, pos, pos_all[k]);
119
120         // Saves all the velocities
121         copy_mat(N_atoms, 3, momentum, vel_all[k]);
122         //But we need to scale the momenta to get the velocities
123         scale_mat(N_atoms, 3, vel_all[k], 1/m_Al);
124     }
125
126     if ((i*10) % N_timesteps == 0){ //Print out progress at every 10%
127         printf("done %d%% of Verlet timestepping\n", (i*10)/N_timesteps);
128     }
129 }
130 printf("done 100%% of Verlet timestepping\n");
131
132 //Calculating MSD
133 printf("calculating MSD\n");
134 get_MSD(N_atoms, N_save_timesteps, pos_all, msd);
135
136 //Calculating the velocity correlation function
137 printf("calculating velocity correlation\n");
138 get_vel_corr(N_atoms, N_save_timesteps, vel_all, vel_corr);
139
140 //Calculating the velocity power spectrum
141 printf("calculating power spectrum\n");
142 get_powerspectrum(N_atoms, N_save_timesteps, vel_all, pow_spec);
143 fft_freq(freq, dt, N_save_timesteps);
144
145 printf("writing to file\n");
146 /* Write tempertaure to file */
147 sprintf(file_name, "../data/temp-%d_pres-%d_Prod-test.tsv",
148         (int) T_eq_C, (int) P_eq_bar);
149 file_pointer = fopen(file_name, "w");
150 for (int i=0; i<N_timesteps; i++){
151     t = i*dt; // time at step i
152     fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n",
153             t, temperature[i], pressure[i]);
154 }
155 fclose(file_pointer);
156
157 /* Write displacements to file */
158 sprintf(file_name, "../data/temp-%d_pres-%d_displacements.tsv",
159         (int) T_eq_C, (int) P_eq_bar);
160 file_pointer = fopen(file_name, "w");
161 for (int i=0; i<N_save_timesteps; i++){
162     t = i*dt*N_between_steps; // time at step i
163     fprintf(file_pointer, "%.4f", t);
164     for (int j=0; j<N_save_atoms; j++){
165         fprintf(file_pointer, "\t %.8f", displacements[i][j]);
166     }
167     fprintf(file_pointer, "\n");
168 }
169 fclose(file_pointer);
170
171 /* Write MSD to file */
172 sprintf(file_name, "../data/temp-%d_pres-%d_dynamicProperties.tsv",
173         (int) T_eq_C, (int) P_eq_bar);
174 file_pointer = fopen(file_name, "w");
175 // write header
176 fprintf(file_pointer, "%s \t [ps] \t MSD[A^2] \t vel_corr [A/ps]^2 \n");
177 for (int i=0; i<N_save_timesteps; i++){
178     t = i*dt*N_between_steps; // time at step i
179     fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n", t, msd[i], vel_corr[i]);
180 }
181 fclose(file_pointer);
182
183 /* Write power spectrum to file */
184 sprintf(file_name, "../data/temp-%d_pres-%d_power-spectrum.tsv",
185         (int) T_eq_C, (int) P_eq_bar);
186 file_pointer = fopen(file_name, "w");
187 // write header

```

```

188 fprintf(file_pointer, "%f f[1/ps] \t P[A/ps]^2 \n");
189 for (int i=0; i<N_save_timesteps/2; i++){ // only save from f=0 to f_crit
190     fprintf(file_pointer, "%.4f \t %.8f \n", freq[i], pow_spec[i]);
191 }
192 fclose(file_pointer);
193
194 // Freeing all the memory
195 free(pos);           pos = NULL;
196 free(pos_0);         pos_0 = NULL;
197 free(momentum);      momentum = NULL;
198 free(forces);         forces = NULL;
199 free(temperature);    temperature = NULL;
200 free(pressure);       pressure = NULL;
201 free(displacements);  displacements = NULL;
202 free(pos_all);        pos_all = NULL;
203 free(vel_all);        vel_all = NULL;
204 free(msd);           msd = NULL;
205 free(vel_corr);       vel_corr = NULL;
206 free(pow_spec);       pow_spec = NULL;
207 free(freq);          freq = NULL;
208
209 return 0;
210 }

```

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

```

1  /*
2   main_Prod.c, Production runs, H1b
3   In this program, we use the equilibrated micro-states from Tasks 3-4 to study
4   dynamical properties, such as mean squared displacement (MSD), velocity
5   auto-correlation function, and the power spectral density of the atom
6   movements.
7
8   System of units:
9   Energy   - eV
10  Time     - ps
11  Length   - Angstrom
12  Temp     - K
13  Mass     - eV (ps)^2 A^(-2)
14  Pressure - eV A^(-3)
15  */
16
17 #include <stdio.h>
18 #include <math.h>
19 #include <stdlib.h>
20 #include <time.h>
21
22 #include "initfcc.h"
23 #include "alpotential.h"
24 #include "funcs.h"
25
26 #define N_cells 4
27 /* define constants in atomic units: eV, , ps, K */
28 #define AMU 1.0364e-4
29 #define degC_to_K 273.15
30 #define bar 6.2415e-07
31 #define kB 8.61733e-5
32
33 /* Main program */
34 int main()
35 {
36     char file_name[100];
37
38     int N_atoms = 4*N_cells*N_cells*N_cells;
39     double m_Al = 27*AMU;
40     double cell_length;
41     double inv_volume;
42
43     double T_eq_C = 500;
44     double P_eq_bar = 1;
45
46     double dt = 5e-4; // higher res for spectral function
47     double t_end = 5;
48     int N_timesteps = t_end/dt;
49     int N_between_steps = 1; // save all steps for max res in spectral function
50     int N_save_timesteps = N_timesteps / N_between_steps;
51     int N_save_atoms = 5;
52
53     double t, E_kin, virial;
54
55     double (*pos)[3] = malloc(sizeof(double[N_atoms][3]));
56     double (*pos_0)[3] = malloc(sizeof(double[N_atoms][3])); //for displacements
57     double (*momentum)[3] = malloc(sizeof(double[N_atoms][3]));
58     double (*forces)[3] = malloc(sizeof(double[N_atoms][3]));
59     double (*displacements)[N_save_atoms] =
60         malloc(sizeof(double[N_save_timesteps][N_save_atoms]));

```

```

61 double (*pos_all)[N_atoms][3] =
62     malloc(sizeof(double)[N_save_timesteps][N_atoms][3]));
63 double (*vel_all)[N_atoms][3] =
64     malloc(sizeof(double)[N_save_timesteps][N_atoms][3]));
65 double *temperature = malloc(sizeof(double)[N_timesteps]);
66 double *pressure = malloc(sizeof(double)[N_timesteps]);
67 double *msd = malloc(sizeof(double)[N_save_timesteps]);
68 double *vel_corr = malloc(sizeof(double)[N_save_timesteps]);
69 double *pow_spec = malloc(sizeof(double)[N_save_timesteps]);
70 double *freq = malloc(sizeof(double)[N_save_timesteps]);
71
72 // Initialize to 0
73 for (int i = 0; i < N_save_timesteps; i++){
74     msd[i] = 0;
75     pow_spec[i] = 0;
76     vel_corr[i] = 0;
77 }
78 FILE *file_pointer;
79
80 // read positions, momenta and cell_length
81 sprintf(file_name, "../data/INIDATA-temp-%d-pres-%d.bin",
82         (int) T_eq_C, (int) P_eq_bar);
83 file_pointer = fopen(file_name, "rb");
84 fread(pos, sizeof(double), 3*N_atoms, file_pointer);
85 fread(momentum, sizeof(double), 3*N_atoms, file_pointer);
86 fread(&cell_length, sizeof(double), 1, file_pointer);
87 fclose(file_pointer);
88
89 for (int i=0; i<N_atoms; i++){
90     for (int j=0; j<3; j++){
91         pos_0[i][j]=pos[i][j];
92     }
93 }
94 inv_volume = pow(N_cells*cell_length, -3);
95 get_forces_AL( forces, pos, cell_length, N_atoms); //initial cond forces
96
97 printf("Initialized. Starting with Verlet timestepping.\n");
98 for (int i=0; i<N_timesteps; i++){
99     /*
100      The loop over the timesteps first takes a timestep according to the
101      Verlet algorithm, then calculates the energies and temeperature.
102     */
103     timestep_Verlet(N_atoms, pos, momentum, forces, m_AL, dt, cell_length);
104
105     E_kin = get_kin_energy(N_atoms, momentum, m_AL );
106     virial = get_virial_AL(pos, cell_length, N_atoms);
107     /* PV = NkT + virial */
108     pressure[i] = inv_volume * (1.5*E_kin + virial);
109     /* 3N*kB*T/2 = 1/(2m) * \sum_{i=1}^N p_i^2 = p_sq/(2m) */
110     temperature[i] = E_kin * 1/(1.5*N_atoms*kB);
111
112     if (i % N_between_steps == 0){
113         int k = i/N_between_steps; // number of saved timesteps so far
114         // Saves the displacements of some atoms into `displacements`
115         get_displacements (N_save_atoms, pos, pos_0, displacements[k]);
116
117         // Saves all the positions
118         copy_mat(N_atoms, 3, pos, pos_all[k]);
119
120         // Saves all the velocities
121         copy_mat(N_atoms, 3, momentum, vel_all[k]);
122         //But we need to scale the momenta to get the velocities
123         scale_mat(N_atoms, 3, vel_all[k], 1/m_AL);
124     }
125
126     if ((i*10) % N_timesteps == 0){ //Print out progress at every 10%
127         printf("done %d0%% of Verlet timestepping\n", (i*10)/N_timesteps);
128     }
129 }
130 printf("done 100%% of Verlet timestepping\n");
131
132 //Calculating MSD
133 printf("calculating MSD\n");
134 get_MSD(N_atoms, N_save_timesteps, pos_all, msd);
135
136 //Calculating the velocity correlation function
137 printf("calculating velocity correlation\n");
138 get_vel_corr(N_atoms, N_save_timesteps, vel_all, vel_corr);
139
140 //Calculating the velocity power spectrum
141 printf("calculating power spectrum\n");
142 get_powerspectrum(N_atoms, N_save_timesteps, vel_all, pow_spec);
143 fft_freq(freq, dt, N_save_timesteps);
144
145 printf("writing to file\n");
146 /* Write temperture to file */
147 sprintf(file_name, "../data/temp-%d-pres-%d-Prod-test.tsv",
148         (int) T_eq_C, (int) P_eq_bar);
149 file_pointer = fopen(file_name, "w");
150 for (int i=0; i<N_timesteps; i++){
151     t = i*dt; // time at step i

```

```

152     fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n",
153             t, temperature[i], pressure[i]);
154 }
155 fclose(file_pointer);
156
157 /* Write displacements to file */
158 sprintf(file_name, "../data/temp-%d_pres-%d_displacements.tsv",
159         (int) T_eq_C, (int) P_eq_bar);
160 file_pointer = fopen(file_name, "w");
161 for (int i=0; i<N_save_timesteps; i++){
162     t = i*dt*N_between_steps; // time at step i
163     fprintf(file_pointer, "%.4f", t);
164     for (int j=0; j<N_save_atoms; j++){
165         fprintf(file_pointer, "\t %.8f", displacements[i][j]);
166     }
167     fprintf(file_pointer, "\n");
168 }
169 fclose(file_pointer);
170
171 /* Write MSD to file */
172 sprintf(file_name, "../data/temp-%d_pres-%d_dynamicProperties.tsv",
173         (int) T_eq_C, (int) P_eq_bar);
174 file_pointer = fopen(file_name, "w");
175 // write header
176 fprintf(file_pointer, "%s t[ps] \t MSD[A^2] \t vel_corr [A/ps]^2 \n");
177 for (int i=0; i<N_save_timesteps; i++){
178     t = i*dt*N_between_steps; // time at step i
179     fprintf(file_pointer, "%.4f \t %.8f \t %.8f \n", t, msd[i], vel_corr[i]);
180 }
181 fclose(file_pointer);
182
183 /* Write power spectrum to file */
184 sprintf(file_name, "../data/temp-%d_pres-%d_power-spectrum.tsv",
185         (int) T_eq_C, (int) P_eq_bar);
186 file_pointer = fopen(file_name, "w");
187 // write header
188 fprintf(file_pointer, "%s f[1/ps] \t P[A/ps]^2 \n");
189 for (int i=0; i<N_save_timesteps/2; i++){ // only save from f=0 to f_crit
190     fprintf(file_pointer, "%.4f \t %.8f \n", freq[i], pow_spec[i]);
191 }
192 fclose(file_pointer);
193
194 // Freeing all the memory
195 free(pos);           pos = NULL;
196 free(pos_0);         pos_0 = NULL;
197 free(momentum);      momentum = NULL;
198 free(forces);        forces = NULL;
199 free(temperature);   temperature = NULL;
200 free(pressure);      pressure = NULL;
201 free(displacements); displacements = NULL;
202 free(pos_all);       pos_all = NULL;
203 free(vel_all);       vel_all = NULL;
204 free(msd);           msd = NULL;
205 free(vel_corr);      vel_corr = NULL;
206 free(pow_spec);      pow_spec = NULL;
207 free(freq);          freq = NULL;
208
209 return 0;
210 }

```

A.6 Misc functions : funcs.c

```

1 #include "funcs.h"
2
3 void add_noise(int M, int N, double mat[M][N], double noise_amplitude )
4 {
5     const gsl_rng_type *T; // static info about rngs
6     gsl_rng *q;           // rng instance
7     gsl_rng_env_setup (); // setup the rngs
8     T = gsl_rng_default;  // specify default rng
9     q = gsl_rng_alloc(T); // allocate default rng
10    gsl_rng_set(q, time(NULL)); // Initialize rng
11
12    // Loops over all the elemtns in the matrix, to which we want to add noise
13    for (int i=0; i<N; i++){
14        for (int j=0; j<M; j++){
15            // adds uniformly distributed random noise in range +/-`noise_amplitude`
16            mat[i][j] += noise_amplitude * (2*gsl_rng_uniform(q)-1);
17        }
18    }
19    gsl_rng_free(q); // deallocate rng
20 }
21
22 void timestep_Verlet ( int N_atoms, double (*pos)[3], double (*momentum)[3],
23                      double (*forces)[3], double m, double dt,
24                      double cell_length){

```

```

25 for (int i = 0; i < N_atoms; i++) {
26     // Half-steps the momentum, then steps the position
27     for (int j = 0; j < 3; j++){
28         // p(t+dt/2)
29         momentum[i][j] += dt * 0.5 * forces[i][j];
30         // q(t+dt)
31         pos[i][j] += dt * momentum[i][j] / m;
32     }
33 }
34 // Updates the forces, based on the new positions
35 // F(t+dt)
36 get_forces_AL( forces, pos, cell_length, N_atoms);
37 // Another half-step in the momenta
38 for (int i = 0; i < N_atoms; i++) {
39     for (int j = 0; j < 3; j++) {
40         // p(t+dt/2)
41         momentum[i][j] += dt * 0.5 * forces[i][j];
42     }
43 }
44 }
45
46 double get_kin_energy ( int N_atoms, double (*momentum)[3], double m ) {
47     double p_sq=0; // momentum squared
48     for (int i = 0; i < N_atoms; i++) {
49         for (int j = 0; j < 3; j++) {
50             p_sq += momentum[i][j] * momentum[i][j];
51         }
52     }
53     // E_kin = p^2/(2m)
54     return p_sq / (2*m);
55 }
56
57 void get_displacements ( int N_atoms, double (*positions)[3],
58                         double (*initial_positions)[3], double disp[]) {
59     for (int i = 0; i < N_atoms; i++) {
60         for (int j = 0; j < 3; j++) {
61             disp[i] += (positions[i][j] - initial_positions[i][j])
62             *(positions[i][j] - initial_positions[i][j]);
63         }
64         disp[i] = sqrt(disp[i]);
65     }
66 }
67
68
69 void get_MSD ( int N_atoms, int N_times, double all_pos[N_times][N_atoms][3],
70               double MSD[N_times]) {
71     // all_pos = positions of all particles at all (saved) times
72     // outer time index it starts at outer it = 1, since MSD[0] = 0
73     for (int it = 1; it < N_times; it++) { //
74         for (int jt = 0; jt < N_times-it; jt++) { // summed time index
75             for (int kn = 0; kn < N_atoms; kn++) { // particle index
76                 for (int kd = 0; kd < 3; kd++) { // three dimensions
77                     MSD[it] += (all_pos[it+jt][kn][kd] - all_pos[jt][kn][kd])
78                     *(all_pos[it+jt][kn][kd] - all_pos[jt][kn][kd]);
79                 }
80             }
81         }
82         MSD[it] *= 1/( (double)N_atoms * (N_times-it));
83     }
84 }
85
86 void get_vel_corr ( int N_atoms, int N_times,
87                    double all_vel[N_times][N_atoms][3],
88                    double vel_corr[N_times]) {
89     /* all_vel = velocity of all particles at all (saved) times */
90     for (int it = 0; it < N_times; it++) { //
91         for (int jt = 0; jt < N_times-it; jt++) { // summed time index
92             for (int kn = 0; kn < N_atoms; kn++) { // particle index
93                 for (int kd = 0; kd < 3; kd++) { // three dimensions
94                     vel_corr[it] += (all_vel[it+jt][kn][kd] * all_vel[jt][kn][kd]);
95                 }
96             }
97         }
98         vel_corr[it] *= 1/( (double)N_atoms * (N_times-it));
99     }
100 }
101
102 void get_powerspectrum ( int N_atoms, int N_times,
103                         double all_vel[N_times][N_atoms][3],
104                         double pow_spec[N_times]) {
105     /* all_vel = velocity of all particles at all (saved) times */
106
107     double vel_component[N_times]; // temp. var. "all_vel[:,i][j]"
108     double pow_spec_component[N_times]; // temp. var. one component F-transf.
109     double normalization_factor = 1/((double) N_atoms*N_times);
110
111     for (int kn = 0; kn < N_atoms; kn++){// for particle index
112         for (int kd = 0; kd < 3; kd++){ // for 3D
113             // Copies the velocity component of one particle, into temporary variable
114             for (int it = 0; it < N_times; it++){
115                 vel_component[it] = all_vel[it][kn][kd];

```



```

116     }
117     //Calculates the power spect. of this one velocity component
118     powerspectrum(vel_component, pow_spec_component, N_times);
119     //Adds the powerspectrum to the "output" variable
120     for (int iw = 0; iw < N_times; iw++) { // for all frequencies
121         pow_spec[iw] += pow_spec_component[iw];
122     } // end for all frequencies
123 } // end for 3D
124 } // end for particle index
125 for (int iw = 0; iw < N_times; iw++) { // for all frequencies
126     pow_spec[iw] *= normalization_factor;
127 }
128 }
129
130
131
132 void copy_mat (int M, int N, double mat_from[M][N], double mat_to[M][N]){
133     /* Copies all matrix elements of `mat_from` to `mat_to` */
134     //loops over all indices
135     for (int i = 0; i < M; i++) {
136         for (int j = 0; j < N; j++) {
137             mat_to[i][j] = mat_from[i][j];
138         }
139     }
140 }
141
142 void set_zero (int M, int N, double mat[M][N]){
143     /* Sets all the elements of matrix `mat` to zero */
144     //loops over all indices
145     for (int i = 0; i < M; i++) {
146         for (int j = 0; j < N; j++) {
147             mat[i][j] = 0;
148         }
149     }
150 }
151
152 void scale_mat (int M, int N, double mat[M][N], double alpha){
153     /* Scales the matrix `mat` by factor `alpha` */
154     //loops over all indices
155     for (int i = 0; i < M; i++) {
156         for (int j = 0; j < N; j++) {
157             mat[i][j] *= alpha;
158         }
159     }
160 }

```

B Auxiliary

B.1 Makefile

```

1
2 CC = gcc
3 CFLAGS = -O3 -Wall -Wno-unused-result
4
5 LIBS = -lm -lgsl -lgslcblas
6
7 HEADERS = initfcc.h alpotential.h funcs.h fft_func.h
8 OBJECTS = initfcc.o alpotential.o funcs.o fft_func.o
9
10
11 %.o: %.c $(HEADERS)
12     $(CC) -c -o $@ $(CFLAGS)
13
14 all: Task1 Task2 Task3 main_Prod.c
15
16 Task1: $(OBJECTS) main_T1.c
17     $(CC) -o $@ $(CFLAGS) $(LIBS)
18
19 Task2: $(OBJECTS) main_T2.c
20     $(CC) -o $@ $(CFLAGS) $(LIBS)
21
22 Task3: $(OBJECTS) main_T3.c
23     $(CC) -o $@ $(CFLAGS) $(LIBS)
24
25 Prod: $(OBJECTS) main_Prod.c
26     $(CC) -o $@ $(CFLAGS) $(LIBS)
27
28 # $(PROGRAMS): $(OBJECTS) main_T1.c
29 #     $(CC) -o $@ $(CFLAGS) $(LIBS)
30
31 clean:
32     rm -f *.o
33     touch *.c

```

C MATLAB scripts

C.1 Analysis scripts for tasks 3-7: A1_energies.m

```
1 %% initial
2
3 tmp = matlab.desktop.editor.getActive; %% cd to current path
4 cd(fileparts(tmp.FileName));
5 set(0,'DefaultFigureWindowSize','docked');
6 warning('off','MATLAB:handle_graphics:exceptions:SceneNode'); % interpreter
7 GRAY = 0.7*[0.9 0.9 1];
8 AMU = 1.0364e-4;
9 m_A1 = 27*AMU;
10 %% task 1: lattice energies
11 clc
12
13 energy_data = load('../data/lattice_energies.tsv');
14 a0 = energy_data(:,1);
15 v0 = a0.^3;
16
17 energy = energy_data(:,2);
18 figure(1);clf;
19 plot(v0,energy, 'xk');
20
21 start_v = 64;
22 end_v = 68;
23 indToInclude = (v0 > start_v) & (v0 < end_v);
24 p = polyfit(v0(indToInclude),energy(indToInclude),2);
25 hold on;
26
27 vvec = linspace(start_v, end_v);
28 plot(vvec, p(1)*vvec.^2 + p(2)*vvec + p(3), '-r');
29 xlim([64 68]);
30
31 v_min = -p(2)/(2*p(1));
32 a_min = v_min^(1/3);
33 omega_res = sqrt(2*p(1)*a_min^4/m_A1);
34 f_res = omega_res/(2*pi); % rough estimation of resonance frequency (?)
35
36 h1 = plot( v_min*[1 1], ax.YLim, '--k'); % plot vertical line at v_min
37
38 ax = gca; ax.YLim = [-13.45 -13.42];
39 ax.YTick = (-13.45:0.01:-13.42);
40 ylabel('$E_{\rm pot}$ [eV/unit cell]');
41 xlabel('$a_0^3$ [\AA^3]');
42 legend('data', 'quadratic fit', ['$V_{\rm eq}$ \approx \, $' ...
43     num2str(round(v_min,2)) '\, \AA^3$', ...
44     'location', 'southeast')
45 ax = gca; ax.Children = ax.Children(3:-1:1);
46 ImproveFigureCompPhys(gcf); h1.LineWidth = 2; setFigureSize(gcf, 300, 600);
47 saveas(gcf, '../figures/potential_energy.eps', 'epsc')
48 %% task 2: find a suitable timestep
49 clc;
50
51 dt=[1e-2,5e-3,2e-3,1e-3];
52 t_eq=0.5;
53
54 figure(1);clf;figure(2);clf;
55
56 for i=1:4
57     T_data = load(sprintf('../data/temperature_dt-%0.0e_Task2.tsv',dt(i)));
58     E_data =load(sprintf('../data/total_energy_dt-%0.0e_Task2.tsv',dt(i)));
59     t = T_data(:,1);
60     T = T_data(:,2);
61     E = E_data(:,2);
62
63     fprintf('dt = %0.0e\n',dt(i));
64
65     T_avg=mean(T(t>t_eq));
66     T_std=std(T(t>t_eq));
67     fprintf('\tT = %0.2f +- %0.1f %%\n', T_avg, abs(T_std/T_avg)*100);
68
69     E_avg=mean(E(t>t_eq));
70     E_std=std(E(t>t_eq));
71     fprintf('\tE = %0.2f +- %0.1f %%\n', E_avg, abs(E_std/E_avg)*100);
72
73     figure(1)
74     plot(t, T); hold on;
75
76     figure(2)
77     plot(t, E);hold on;
78 end
79 for ifig = 1:2
80     figure(ifig);
81     h = legend(strcat({'$dt = $ '}, num2str(round(dt,4)) , ' ps'));
82     xlabel('$t$ [ps]');
83     ax = gca;
```

```

84     if ifig ==1
85         ylabel('$T$ [K]')
86         ax.YLim = [400 1800];
87     else
88         ylabel('$E_{\rm tot}$ [eV/unit cell]');
89         ax.YTick = (-13:0.1:-10);
90         ax.YLim = [-12.6 -12.0];
91     end
92     ImproveFigureCompPhys(gcf,'Linewidth', 2);setFigureSize(gcf, 400, 400);
93 end
94 saveas(1, '../figures/dt-scan-temperature.eps', 'epsc')
95 saveas(2, '../figures/dt-scan-energy.eps', 'epsc')
96 %% task 3: temperature and pressure equilibration,
97 % and task4: test production pressure and temperature
98
99 clc; clf;
100 temps = [500 700 500 700];
101 temperatures_str = num2str([500;700]);
102 FILENAMES = [strcat({'../data/temp-'}, temperatures_str,...
103     '_pres-1_Task3.tsv');
104     strcat({'../data/temp-'}, temperatures_str, '_pres-1_Prod-test.tsv')];
105 bar = 6.2415e-07;
106 Kelvin_to_degC = -273.15;
107 t_eqs = [1 1 0.5 0.5]; % approximate equilibration time
108 N_average_points = 50;
109 dt = 5e-3;
110 tau_eqilibration = 100*dt;
111
112 for iFile = 1:numel(FILENAMES)
113     figure(iFile);clf;
114     data = load(FILENAMES{iFile});
115
116     t = data(:,1);
117     T = data(:,2)+Kelvin_to_degC;
118     P = data(:,3)/bar;
119
120     t_eq=t_eqs(iFile);
121
122     %fprintf('dt = %0.0e\n',dt(i));
123     T_avg=mean(T(t>t_eq));
124     T_std=std(T(t>t_eq));
125     fprintf('\tT = %0.2f +- %0.1f K\n', T_avg, abs(T_std));
126
127     P_avg=mean(P(t>t_eq));
128     P_std=std(P(t>t_eq));
129     fprintf('\tP = %0.2f +- %0.1f bar\n', P_avg, abs(P_std));
130
131     yyaxis left
132
133     if iFile <=2 % equilibration run, otherwise production
134         plot(t./tau_eqilibration,T, 'color', GRAY),hold on;
135         plot(t./tau_eqilibration, movmean(T,N_average_points),'-k')
136     else
137         plot(t,T, 'color', GRAY),hold on;
138         plot(t, cumsum(T)./(1:length(t)),'-k')
139     end
140     ylabel('$T \, [\mathrm{K}]$')
141
142     if iFile <=2 % equilibration run, otherwise production
143         ylim(temps(iFile)*(1+ 0.3*[-1,1.2]))
144         yyaxis right
145         plot(t./tau_eqilibration,P),hold on;
146         plot(t./tau_eqilibration, movmean(P,N_average_points),'-k')
147         legend('$\mathcal{T}$', 'mov avg', '$\mathcal{P}$', 'mov avg');
148         xlabel('$t/\tau_{\rm eq}$')
149         %xlim([0 5])
150     else
151         ylim(temps(iFile)+ 100*[-3,3])
152         yyaxis right
153         plot(t,P),hold on;
154         plot(t, cumsum(P)./(1:length(t)),'-k')
155         legend('$\mathcal{T}$', 'cum avg', '$\mathcal{P}$', 'cum avg');
156         xlabel('$t \, [\mathrm{ps}]$')
157     end
158     ylabel('$P \, [\mathrm{bar}]$')
159     ylim([-100,400])
160     ImproveFigureCompPhys(gcf, 'linewidth', 3, 'LineColor', ...
161         {'MYORANGE', GRAY, 'MYBLUE', GRAY});
162     setFigureSize(gcf, 400, 400);
163 end
164
165 saveas(1, '../figures/TP-eq-500.eps', 'epsc')
166 saveas(2, '../figures/TP-eq-700.eps', 'epsc')
167 saveas(3, '../figures/TP-prod-500.eps', 'epsc')
168 saveas(4, '../figures/TP-prod-700.eps', 'epsc')
169 %% determine displacements and MSD
170 temperatures_str = num2str([500;700]);
171 clc; clf;
172 figure(10); clf;
173 FILENAMES = strcat({'../data/temp-'}, temperatures_str, ...
174     '_pres-1_displacements.tsv');

```

```

175 FILENAMES_Dyn = strcat({'../data/temp-'}, temperatures_str, ...
176 '_pres-1_dynamicProperties.tsv');
177 FILENAMES_Pow = strcat({'../data/temp-'}, temperatures_str, ...
178 '_pres-1_power-spectrum.tsv');
179 for iFile = 1:numel(FILENAMES)
180     figure(iFile); clf;
181     data = load(FILENAMES{iFile});
182     t = data(:,1);
183     dx = data(:,2:end);
184
185     data = load(FILENAMES_Dyn{iFile});
186     MSD = data(:,2);
187     vel_corr = data(:,3);
188     plot(t, MSD, 'k'); hold on;
189
190     if iFile == 2 % liquid
191         tStart = 1;
192         D = MSD(t>tStart)./(6*t(t>tStart));
193         selfDiffusionCoeff = mean(D); % in  $\text{\AA}^2/\text{ps}$ 
194         plot(t, 6*t*selfDiffusionCoeff, 'r');
195     end
196
197     plot(t, dx.^2, 'color', GRAY); hold on;
198
199     xlabel('$t$ [ps]')
200     ylabel('$\Delta x^2$, [\rm \AA^2]$')
201     if iFile == 1
202         ylim([0 1.0]);
203         leg = legend('$\Delta_{\rm MSD}$', 'individual trajectories');
204     else
205         ylim([0 20]);
206         leg = legend('$\Delta_{\rm MSD}$', '$6 t D_s$', ...
207             'individual trajectories');
208     end
209
210     leg.Location='northwest';
211     ImproveFigureCompPhys(gcf, 'Linewidth', 2);
212     ax = gca; [ax.Children(6:end).LineWidth] = deal(5);
213     ax.Children = ax.Children([6:end 1:5]);
214     setFigureSize(gcf, 400, 400);
215 end
216
217 % velocity correlation
218 figure(10);clf; figure(11);clf;
219 n_average_points = 1;%30;
220 for iFile = 1:numel(FILENAMES)
221     data = load(FILENAMES_Dyn{iFile});
222     t = data(:,1);
223     vel_corr = data(:,3);
224
225     data = load(FILENAMES_Pow{iFile});
226     freq = data(:,1);
227     pow_spec = data(:,2);
228
229     figure(10);
230     plot(t, vel_corr/vel_corr(1)); hold on;
231
232     dt = t(2)-t(1);
233     N_times = round(length(t)/2);% we have bad statistics at later times.
234     deltaf = 1/(N_times * dt);
235     freqvec = 0:deltaf:(1/(2*dt));
236     PhiHat = 2 * trapz(t(1:N_times), ...
237         (vel_corr(1:N_times) * ones(size(freqvec))) .* ...
238         cos(2*pi*t(1:N_times) * freqvec), 1);
239
240     figure(11);
241     plot(freqvec, m_Al/2*PhiHat); hold on;
242     plot(freq, m_Al/2* pow_spec*t(end), ':'); hold on;
243     if iFile == 2 % liquid
244         tStart = 1;
245         selfDiffusionCoeff_spectral = PhiHat(1)/6; % in  $\text{\AA}^2/\text{ps}$ 
246     end
247 end
248
249 disp([selfDiffusionCoeff selfDiffusionCoeff_spectral]);
250
251 figure(10)
252 xlim([0 1]);
253 leg = legend(strcat({'$T=$'}, num2str([500;700]), '\, $\circ$ $C$'));
254 leg.Location='northeast';
255 xlabel('$t$ [ps]')
256 ylabel('$\Phi(t)/\Phi(0)$')
257 ImproveFigureCompPhys(gcf, 'LineColor', {'MYRED', 'MYLIGHTBLUE'});
258 setFigureSize(gcf, 400, 400);
259
260 figure(11)
261 leg = legend('$T= 500 \, \circ \, C, \hat{\Phi}$', ...
262     '$T= 500 \, \circ \, C, \hat{P}$', ...
263     '$T= 700 \, \circ \, C, \hat{\Phi}$', ...
264     '$T= 700 \, \circ \, C, \hat{P}$');

```

```

266 xlim([0 30])
267 ylim([0 Inf])
268 xlabel('$f$ [THz]')
269 ylabel('$\frac{1}{2} m \hat{P}$ [eV/THz]')
270 setFigureSize(gcf, 400, 400);
271
272 ImproveFigureCompPhys(gcf,'LineColor', ...
273     {'r', 'MYRED', 'GERIBLUE','MYLIGHTBLUE'}');
274 saveas(1, '../figures/MSD-500.eps', 'epsc')
275 saveas(2, '../figures/MSD-700.eps', 'epsc')
276 saveas(10, '../figures/Phi-t.eps', 'epsc')
277 saveas(11, '../figures/P-freq.eps', 'epsc')

```

C.2 Improve figure appearance: ImproveFigureCompPhys.m

```

1 function ImproveFigureCompPhys(varargin)
2 %ImproveFigureCompPhys Improves the figures of supplied handles
3 % Input:
4 % - none (improve all figures) or handles to figures to improve
5 % - optional:
6 %     LineWidth int
7 %     LineStyle column vector cell, e.g. {'-','--'}',
8 %     LineColor column vector cell, e.g. {'k',[0 1 1], 'MYBLUE'}'
9 %             colors: MYBLUE,MYORANGE,MYGREEN,MYPURPLE, MYYELLOW,
10 %             MYLIGHTBLUE, MYRED
11 %     Marker column vector cell, e.g. {'.', 'o', 'x'}'
12
13 % ImproveFigure was originally written by Adam Stahl, but has been heavily
14 % modified by Linnea Hesslow
15
16
17 %% Handle inputs
18 % If no inputs or if the first argument is a string (a property rather than
19 % a handle), use all open figures
20 if nargin == 0 || ischar(varargin{1})
21     %Get all open figures
22     figHs = findobj('Type','figure');
23     nFigs = length(figHs);
24 else
25     % Check the supplied figure handles
26     figHs = varargin{1};
27     figHs = figHs(ishandle(figHs) == 1); %Keep only those handles that are ←
28     % proper graphics handles
29     nFigs = length(figHs);
30 end
31
32 % Define desired properties
33 titleSize = 24;
34 interpreter = 'latex';
35 lineWidth = 4;
36 axesWidth = 1.5;
37 labelSize = 22;
38 textSize = 20;
39 legTextSize = 18;
40 tickLabelSize = 18;
41 LineColor = {};
42 LineStyle = {};
43 Marker = {};
44
45 % define colors
46 co = [ 0      0.4470    0.7410
47       0.8500    0.3250    0.0980
48       0.9290    0.6940    0.1250
49       0.4940    0.1840    0.5560
50       0.4660    0.6740    0.1880
51       0.3010    0.7450    0.9330
52       0.6350    0.0780    0.1840 ];
53 colors = struct('MYBLUE', co(1,:),...
54     'MYORANGE', co(2,:),...
55     'MYYELLOW', co(3,:),...
56     'MYPURPLE', co(4,:),...
57     'MYGREEN', co(5,:),...
58     'MYLIGHTBLUE', co(6,:),...
59     'MYRED',co(7,:),...
60     'GERIBLUE', [0.3000    0.1500    0.7500],...
61     'GERIRET', [1.0000    0.2500    0.1500],...
62     'GERIYELLOW', [0.9000    0.7500    0.1000],...
63     'LIGHTGREEN', [0.4    0.85    0.4],...
64     'LINNEAGREEN', [7 184 4]/255);
65
66 % Loop through the supplied arguments and check for properties to set.
67 for i = 1:nargin
68     if ischar(varargin{i})
69         switch lower(varargin{i}) %Compare lower case strings
70             case 'linewidth'
71                 lineWidth = varargin{i+1};

```

```

71         case 'linestyle'
72             LineStyle = varargin{i+1};
73         case 'linecolor'
74             LineColor = varargin{i+1};
75             for iLineColor = 1:numel(LineColor)
76                 if isfield(colors, LineColor{iLineColor})
77                     LineColor{iLineColor} = colors.(LineColor{iLineColor});
78                 end
79             end
80         case 'marker'
81             Marker = varargin{i+1};
82         end
83     end
84 end
85 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
86
87 %%% Improve the figure(s)
88
89 for iFig = 1:nFigs
90     fig = figHs(iFig);
91
92     lineObjects = findall(fig, 'Type', 'line');
93     textObjects = findall(fig, 'Type', 'text');
94     axesObjects = findall(fig, 'Type', 'axes');
95     legObjects = findall(fig, 'Type', 'legend');
96     contourObjects = findall(fig, 'Type', 'contour'); % not counted as lines
97
98     %%% TEXT APPEARANCE: first set all to textSize and then change the ones
99     %%% that need to be changed again
100
101     %Change size of any text objects in the plot
102     set(textObjects, 'FontSize', textSize);
103     set(legObjects, 'FontSize', legTextSize);
104
105     %%% FIX LINESTYLE, COLOR ETC. FOR EACH PLOT SEPARATELY
106     for iAx = 1:numel(axesObjects)
107         lineObjInAx = findall(axesObjects(iAx), 'Type', 'line');
108
109         %set line style and color style (only works if all figs have some
110         %number of line plots..)
111         if ~isempty(LineStyle)
112             set(lineObjInAx, {'LineStyle'}, LineStyle)
113             set(contourObjects, {'LineStyle'}, LineStyle); %%%
114         end
115         if ~isempty(LineColor)
116             set(lineObjInAx, {'Color'}, LineColor)
117             set(contourObjects, {'LineColor'}, LineColor); %%%
118         end
119         if ~isempty(Marker)
120             set(lineObjInAx, {'Marker'}, Marker)
121             set(lineObjInAx, {'Markersize'}, num2cell(10+22*strcmp(Marker, '.'))←
122                 )
123         end
124
125         %%% change font sizes.
126         % Tick label size
127         xLim = axesObjects(iAx).XLim;
128         axesObjects(iAx).FontSize = tickLabelSize;
129         axesObjects(iAx).XLim = xLim;
130         %Change label size
131         axesObjects(iAx).XLabel.FontSize = labelSize;
132         axesObjects(iAx).YLabel.FontSize = labelSize;
133
134         %Change title size
135         axesObjects(iAx).Title.FontSize = titleSize;
136     end
137
138     %%% LINE APPEARANCE
139     %Change line thicknesses
140     set(lineObjects, 'LineWidth', lineWidth);
141     set(contourObjects, 'LineWidth', lineWidth);
142     set(axesObjects, 'LineWidth', axesWidth)
143
144     % set interpreter: latex or tex
145     set(textObjects, 'interpreter', interpreter)
146     set(legObjects, 'Interpreter', interpreter)
147     set(axesObjects, 'TickLabelInterpreter', interpreter);
148 end
149 end

```

C.3 Change size of figures: setFigureSize.m

```

1 function [ fig ] = setFigureSize( fig, H, W )
2 fig.Units = 'points';
3 fig.WindowStyle = 'normal'; % undock

```

```
4 fig.Position(3:4) = [W H];  
5 end
```