Simulation Methods in Physics I

Worksheet 2: Statistical Mechanics and Molecular Dynamics

Students:	Michael Marquardt	Cameron Stewart
matriculation numbers:	3122118	3216338

1 Statistical Mechanics

1.1 Configurations of Thermodynamic Systems

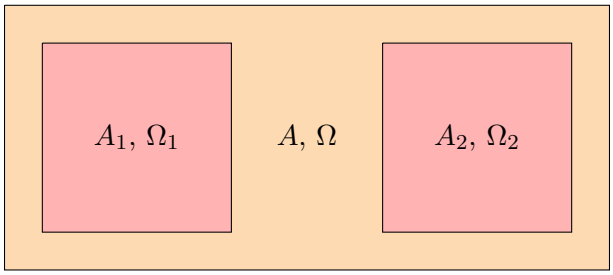

Fig. 1: Thermodynamic System A with a Ω possible configurations which is made up of the two Systems A_1 and A_2.

Figure 1 shows a thermodynamic System A which consists of two subsystems A_1 and A_2. Thereby the system A_1 has $\Omega_1 = 10^{31}$ and A_2 $\Omega_2 = 10^{28}$ possible configurations. The number of possible configurations Ω of the combined system A therefore is the product of Ω_1 and Ω_2:

$$\Omega = \Omega_1 \cdot \Omega_2 = 10^{31+28} = 10^{59} \tag{1}$$

The entropy of such a system is $S = k_B \ln \Omega$, where $k_B = 1.38 \cdot 10^{-23}\,\mathrm{J\,K^{-1}}$ is the Boltzmann constant. Therefor the entropies of A, A_1 and A_2 are:

$$S_1 = k_B \cdot \ln \Omega_1 = 31 k_B \cdot \ln 10 = 9.850 \cdot 10^{-22}\,\mathrm{J\,K^{-1}} \tag{2}$$

$$S_2 = k_B \cdot \ln \Omega_2 = 28 k_B \cdot \ln 10 = 8.897 \cdot 10^{-22}\,\mathrm{J\,K^{-1}} \tag{3}$$

$$\begin{aligned} S &= k_B \cdot \ln \Omega = k_B \cdot \ln (\Omega_1 \cdot \Omega_2) = k_B \cdot \ln \Omega_1 + k_B \cdot \ln \Omega_2 \\ &= S_1 + S_2 = 59 k_B \cdot \ln 10 = 18.748 \cdot 10^{-22}\,\mathrm{J\,K^{-1}} \end{aligned} \tag{4}$$

As you can see, the entropy of the combined system is the sum of its sybsystems entropies.

1.2 Isothermal Change of States

Consider a volume $V_0 = 1\,\mathrm{m}^3$ of the nearly ideal gas neon at a pressure $p_0 = 1\,\mathrm{atm} = 1013.25\,\mathrm{hPa}$ and temperature $T = 298\,\mathrm{K}$. Now the Volume of this gas is expanded isothermal by $1\,\% = f_V - 1$. For an ideal 1-atomic gas the intrinsic energy is $U = \frac{3}{2} N k_B T$. Because of this U does not change through an isothermal process. At least we can use the first principle of thermodynamics (5) and the ideal gas law (6) in order to derive the the factor f_Ω by which the number of possible states Ω increase.

$$\Delta U = \Delta Q + \Delta W \tag{5}$$
$$pV = Nk_BT \tag{6}$$
$$\Delta T = 0 \quad \Rightarrow \quad \Delta U = \frac{3}{2} Nk_B \cdot \Delta T = 0 \tag{7}$$
$$\mathrm{d}S = \frac{\mathrm{d}Q}{T} \overset{(5)}{=} \frac{\mathrm{d}U - \mathrm{d}W}{T} \overset{(7)}{=} \frac{p}{T}\mathrm{d}V \overset{(6)}{=} \frac{Nk_B}{V}\mathrm{d}V \qquad |\int \tag{8}$$
$$\int_{S_0}^{S_1} \mathrm{d}S = \int_{V_0}^{V_1} \frac{Nk_B}{V}\mathrm{d}V \tag{9}$$
$$\Delta S = Nk_B \cdot (\ln(V_1) - \ln(V_0)) = \frac{p_0 V_0}{T} \cdot \ln\left(\frac{V_0 + \Delta V}{V_0}\right) \tag{10}$$
$$k_B \cdot \ln\left(\frac{\Omega_0 + \Delta\Omega}{\Omega_0}\right) = \frac{p_0 V_0}{T} \cdot \ln\left(\frac{V_0 + \Delta V}{V_0}\right) \qquad |:k_B \tag{11}$$
$$\ln(f_\Omega) = \frac{p_0 V_0}{k_B T} \cdot \ln(f_V) = \ln\left(f_V^{\frac{p_0 V_0}{k_B T}}\right) \qquad |\exp \tag{12}$$
$$f_\Omega = f_V^{\frac{p_0 V_0}{k_B T}} = f_V^N = 29.5^{1/k_B} \rightarrow \text{overflow} \tag{13}$$

This is an incredible high number. This shall show that only a little more space for our simulation increases the number of possible staes with the power $N = \frac{p_0 V_0}{k_B T} \approx 41\,\text{mol}$ (number of particles).

1.3 Isochoral Change of States

The considered system contains $N = 1\,\text{mol}$ of particles at a temperature of $T_0 = 400\,\text{K}$. Now an energy of $\Delta U = 100\,\text{kJ}$ is added to the system. We again want to figure out f_Ω.

$$\Delta V = 0 \quad \Rightarrow \quad \Delta W = -p\Delta V = 0 \tag{14}$$
$$\mathrm{d}S = \frac{\mathrm{d}Q}{T} \overset{(5)}{=} \frac{\mathrm{d}U}{T} \tag{15}$$
$$\mathrm{d}U = \frac{3}{2} k_B N \cdot \mathrm{d}T \quad \text{(ideal gas)} \tag{16}$$
$$f_T = \frac{T_0 + \Delta T}{T_0} = 1 + \frac{2\Delta U}{3k_B N T_0} \tag{17}$$
$$= 21.06 \tag{18}$$
$$\mathrm{d}S = \frac{3k_B N}{2T}\mathrm{d}T \qquad |\int \tag{19}$$
$$\int_{S_0}^{S_1} \mathrm{d}S = \int_{T_0}^{T_1} \frac{3k_B N}{T}\mathrm{d}T \tag{20}$$
$$k_B \cdot \ln(f_\Omega) = \frac{3}{2} k_B N \cdot (\ln(T_1) - \ln(T_0)) = \frac{3}{2} k_B N \cdot \ln(f_T) \qquad |:k_B \,|\exp \tag{21}$$
$$f_\Omega = f_T^{\frac{3}{2}N} = 21.06^{\frac{3}{2}\text{mol}} \rightarrow \text{overflow} \tag{22}$$

As you can see, this number is also very large. The number of possible states increases with a factor to the power $\frac{3}{2}N = \frac{3}{2}\text{mol}$.

1.4 Thermodynamic Variables in the Canonical Ensemble

Helmholtz free energy:

$$F = U\left(S(N,V,T),V,N\right) - T \cdot S\left(T,V,N\right) = -k_B T \cdot \ln\left(Z(T,V,N)\right) \tag{23}$$

From equation (23) we can derive the intrinsic energy U, the pressure p and the entropy S. Therefor we use $\frac{\partial U}{\partial S} = T$.

$$\frac{\partial F}{\partial V} = \frac{\partial U}{\partial S}\frac{\partial S}{\partial V} + \frac{\partial U}{\partial V} - T\frac{\partial S}{\partial V} = -p(T,V,N) \tag{24}$$

$$p(T,V,N) = \frac{k_B T}{Z(T,V,N)} \cdot \frac{\partial Z}{\partial V}(T,V,N) \tag{25}$$

$$\frac{\partial F}{\partial T} = \frac{\partial U}{\partial S}\frac{\partial S}{\partial T} - T\frac{\partial S}{\partial T} - S = -S(T,V,N) \tag{26}$$

$$S(T,V,N) = \frac{k_B T}{Z(T,V,N)} \cdot \frac{\partial Z}{\partial T}(T,V,N) \tag{27}$$

$$U(T,V,N) = T \cdot S(T,V,N) + F(T,V,N) \tag{28}$$

$$= T \cdot S(T,V,N) - k_B T \cdot \ln\left(Z(T,V,N)\right) \tag{29}$$

1.5 Ideal Gas

The canonical partition function if an ideal gas is:

$$Z(T,V,N) = \frac{V^N}{\lambda^{3N} N!} \qquad \lambda = \sqrt{\frac{h^2 \beta}{2\pi m}} \qquad \beta = \frac{1}{k_B T} \tag{30}$$

Now we calculate the Helmholtz free energy F (23) for this partition function Z.

$$F(T,V,N) = -k_B T \cdot \ln\left(\frac{V^N}{\lambda^{3N} N!}\right) \tag{31}$$

$$= -k_B T \cdot \left(N \cdot \ln\left(\frac{V}{\lambda^3}\right) - \ln\left(N!\right)\right) \tag{32}$$

$$\overset{N \geq 10^3}{=} -k_B T N \cdot \left(\ln\left(\frac{V}{\lambda^3 N}\right) + 1\right) \tag{33}$$

$$= -k_B T N \cdot \left(\ln\left(\frac{V}{N}\left(\frac{2\pi m k_B T}{h^2}\right)^{3/2}\right) + 1\right) \tag{34}$$

The last step is to derive an expression for the heat capacity at constant volume C_V.

$$C_V = \left(\frac{\partial Q}{\partial T}\right)_V = \left(\frac{\partial U}{\partial T}\right)_V \overset{(28)}{=} \left(\frac{\partial (T \cdot S + F)}{\partial T}\right)_V \tag{35}$$

$$S(T,V,N) \overset{(27)}{=} -\frac{\partial F}{\partial T} \tag{36}$$

$$\overset{(34)}{=} \frac{\partial}{\partial T} k_B T N \cdot \left(\ln\left(\frac{V}{N}\left(\frac{2\pi m k_B T}{h^2}\right)^{3/2}\right) + 1\right) \tag{37}$$

$$= k_B N \cdot (\ln(\ldots) + 1) + \frac{3}{2} k_B N \cdot \left(\frac{2\pi m k_B T}{h^2}\right)^{3/2} \cdot \left(\frac{2\pi m k_B T}{h^2}\right)^{-3/2} \tag{38}$$

$$= -\frac{F}{T} + \frac{3}{2} k_B N \tag{39}$$

$$C_V \overset{(39)}{=} \left(\frac{\partial \left(-T \cdot \frac{F}{T} + \frac{3}{2} k_B N T + F\right)}{\partial T}\right)_V = \left(\frac{\partial}{\partial T} \frac{3}{2} k_B N T\right)_V \tag{40}$$

$$= \frac{3}{2} k_B N \tag{41}$$

2 Molecular Dynamics: Lennard-Jones Fluid

In this worksheet we will simulate a simple Lennard-Jones fluid using Molecular Dynamics. We begin by implementing the Lennard-Jones (LJ) potential and force in python and simulate a few particles as "billards". We will move on to simulating a LJ fluid using periodic boundary conditions and random initial velocities. We then compare the run times for this system using python, C, and cython. Finally, we include cell lists and Verlet lists in order to further improve our efficiency.

2.1 Lennard Jones Potential

The LJ potential is originally a model used to represent a system of noble gas particles and consists of a hard core repulsion as well as a longer ranged attractive potential due to the dipole-dipole interaction. At longer ranges the potential is nearly zero. It is given by

$$V_{\mathrm{LJ}}(r) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right). \tag{42}$$

2.2 Reduced Units

We may define a set of dimensionless reduced units because the LJ potential is of the general form

$$U(r) = \epsilon \phi(\sigma / r) \tag{43}$$

where σ and ϵ are constants and ϕ is smooth and differentiable. We use

$$x^* = x/\sigma, \quad V^* = V/\sigma^3, \quad T^* = k_{\mathrm{B}} T/\epsilon, \quad p^* = p\sigma^3/\epsilon. \tag{44}$$

The law of corresponding states claims that any system interacting with an potential, such as (43), is described by the same equation of state. We may therefore simulate at $\epsilon = \sigma = 1.0$ without any loss of generality.

With the help of numpy, we implement the LJ potential in the following way:

Code block 1: LJ Potential — script: src/ljlib.py

```
10 def compute_lj_potential(r_ij):
11     """Compute LJ potential on particle j from particle i
           """
12     d = la.norm(r_ij)
13     return 4*(d**-12 - d**-6)
```

The force due to this potential is simply given by the negative gradient and is implemented as:

Code block 2: LJ Force — script: src/ljlib.py

```
15 def compute_lj_force(r_ij):
16     """Compute LJ force on particle j due to particle i"""
17     d = la.norm(r_ij)
18     f = 48*(d**-13 - 0.5*(d**-7))
19     return f*r_ij/d
```

The output of these functions for distances from 0.85 to 2.5 can be seen in Fig. 2.

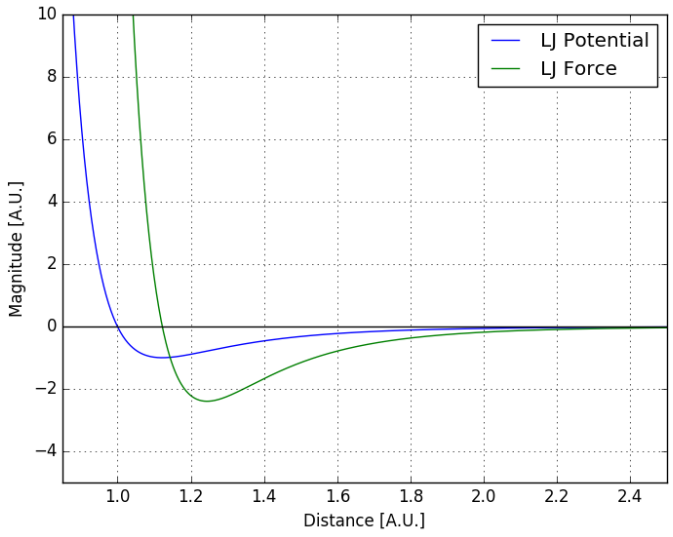

Fig. 2: Output of the compute_lj_potential and compute_lj_force functions.

2.3 Lennard-Jones Billards

We would now like to use this potential to run an MD simulation. We use the velocity verlet integrator to run the MD. Our first simulation will place 5 particles by hand and integrate until time $t = 20.0$ with a timestep of $dt = 0.01$. We begin with a supplied template and simply import the library created in the previous task.

Code block 3: Import ljlib.py script: src/ljbillards.py

```
1 from numpy import *
2 from matplotlib.pyplot import *
3 from ljlib import *
```

We can use VMD in order to visualize the simulation once we've run the program. A still shot from our simulation can be seen in Fig. 3. In the original simulation, the second

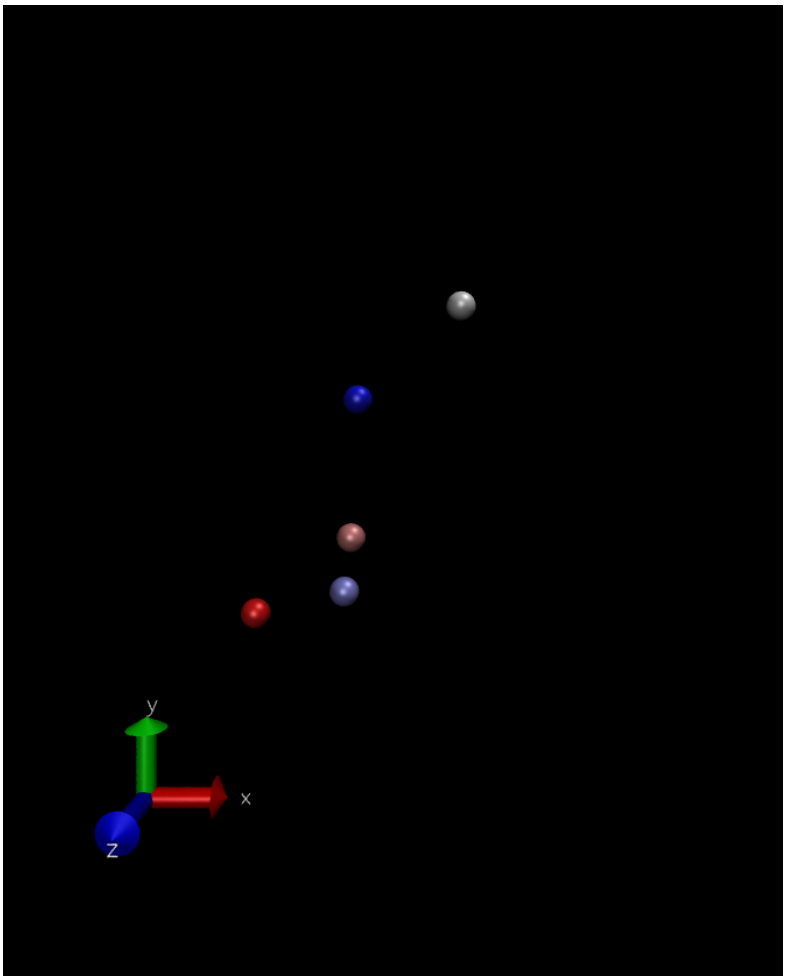

Fig. 3: A snapshot of the LJ billards simulation.

particle (for indices look at Fig. 4) did not collide with the fourth. We moved the fourth particle slightly south in order to correct this. The new trajectory can be seen in Fig. 4. Another feature of this trajectory is that the first and third particles do not behave like billard balls at all. This is due to the attractive part of the LJ potential.

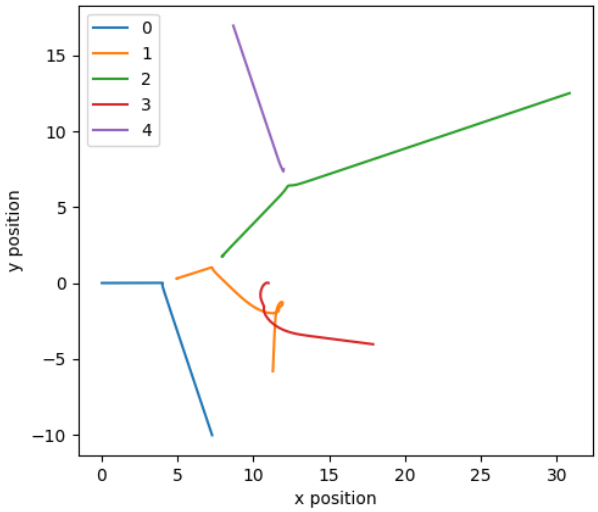

Fig. 4: Billard trajectory with modified initial conditions.

Furthermore we can take a look at the total energy of the simulated system in Fig 5. As you can see the energy varies heavily when two particles collide, but after that the energy quickli returns to its former value. All in all the energy stays constant over time just like we expected it from the algorithm.

2.4 Periodic Boundary Conditions

Because computers have limited resources and computing power, we can never simulate a full system. Rather we simulate a much smaller system. Unfortunately, if we had only a few particles in a boundless space, they would escape to infinity and no longer interact. We can't simply add walls because we must then account for surface effects. We can solve this problem by using periodic boundary conditions (PBC). Images of our central box essentially extend infinitely in every direction. We now have an infinite volume V and particle number N but we may prescribe a density $\rho = N/V$.

2.4.1 Long and Short Range Interactions

One problem with an infinite number of particles is that long range interactions become hard to model since we must compute an infinite number of interactions. Fortunately, LJ is essentially short ranged and we can define a cutoff distance as follows and implent this for both the LJ potential and force (Code Block 4).

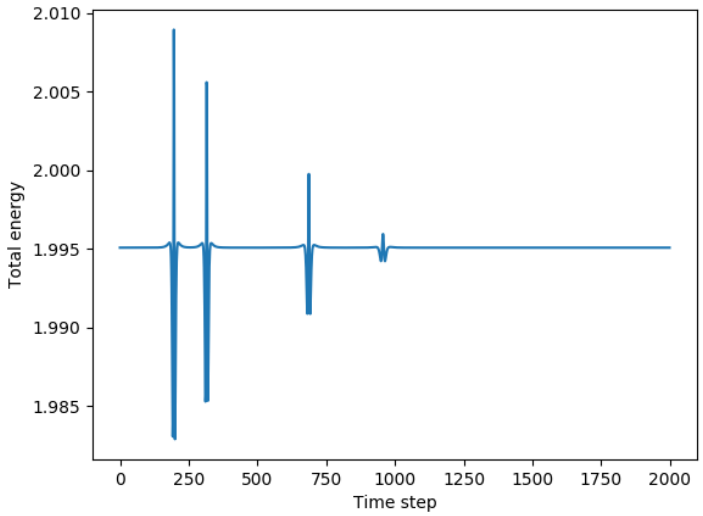

Fig. 5: Energy of the system with modified initial conditions over time.

$$V'_{\mathrm{LJ}}(r) = \begin{cases} V_{\mathrm{LJ}}(r) - V_{\mathrm{LJ}}(r_{\mathrm{cutoff}}), & r \leq r_{\mathrm{cutoff}} \\ 0, & \text{else} \end{cases}$$

Code block 4: LJ Cutoff — script: src/ljlib.py

```
21 def compute_cutoff_potential(r_ij):
22     """Compute cutoff potential on particle j from
           particle i"""
23     d = la.norm(r_ij)
24     if d <= 2.5:
25         return compute_lj_potential(r_ij) -
               compute_lj_potential(np.array([2.5,0,0]))
26     else:
27         return 0
28
29 def compute_cutoff_force(r_ij):
30     """compute cutoff force on particle j from particle i
           """
31     d = la.norm(r_ij)
32     if d <= 2.5:
33         return 48*(d**-13 - 0.5*d**-7)*r_ij/d
34     else:
35         return 0
```

We use a value of 2.5 for r_{cutoff}.

2.4.2 Minimum Image Convention

If we use a cutoff range that is less than the length of our box L then we only have to compute the interaction between the closest image of the particles. All other images will be too far to interact. Knowing this, we can implement periodic boundary conditions by simply calculating the minimum image distance and using that distance to calculate the forces. Our particles will then freely propogate out of the box while still experiencing the correct force. When looking at the trajectory in VMD or in python we can then simpy wrap the particle locations back to the first box to see the true trajectories. The following code block shows how the minimum image is calculated:

Code block 5: Minimum Image — script: src/ljbillards1.py

```
54 def image(rij):
55     """Return the minimum image of particle j for particle
           i"""
56     rij -= L*rint(rij/L)
57     return rij
```

Thereby the length L is defined elsewhere in the script as $L = 10.0$. The compute_forces function was modified as well in order to include the functions image and the compute_cutoff_force. The function compute_energy is extended in the same way.

Code block 6: PBC Forces — script: src/ljbillards1.py

```
 5 def compute_forces(x):
 6     """Compute and return the forces acting onto the
           particles,
 7     depending on the positions x."""
 8     global epsilon, sigma
 9     _, N = x.shape
10     f = zeros_like(x)
11     for i in range(1,N):
12         for j in range(i):
13             # distance vector
14             rij = x[:,j] - x[:,i]
15             rij = image(rij)
16             fij = compute_cutoff_force(rij)
17             f[:,i] -= fij
18             f[:,j] += fij
19     return f
```

We tested the periodic boundary conditions by setting up two particles with the following initial conditions:

Code block 7: Initial Conditions script: src/ljbillards1.py

```
67 # particle positions
68 x = zeros((3,2))
69 x[:,0] = [3.9, 3.0, 4.0]
70 x[:,1] = [6.1, 5.0, 6.0]
71
72 # particle velocities
73 v = zeros((3,2))
74 v[:,0] = [-2.0, -2.0,-2.0]
75 v[:,1] = [2.0, 2.0, 2.0]
```

The trajectory can be seen in Fig. 6.

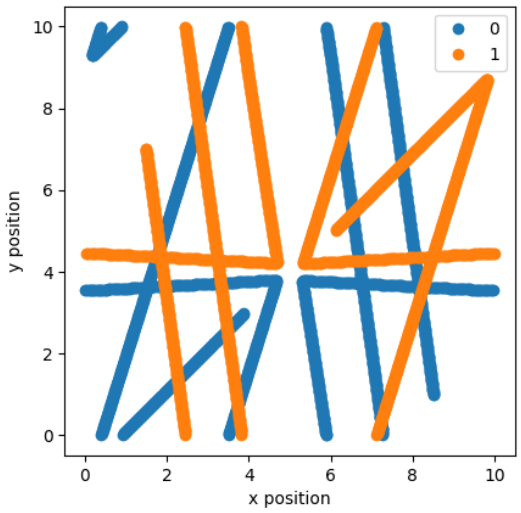

Fig. 6: Billard trajectory with periodic boundary conditions for two particles (0 and 1).

2.5 Lennard-Jones Fluid

2.5.1 Pure Python

We now inherit a nice python program to simulate an LJ fluid with periodic boundary conditions. We have only to import `ljlib.py`, compute the size of the system, and place the particles programatically as shown in the following code block.

Code block 8: Particle Placement script: src/ljfluid.py

```
81 # - compute the size of the system
82 L = (N/density)**(1./3.)
```

```
83
84 # - set up n*n*n particles on a cubic lattice
85 for i in range(n):
86     for j in range(n):
87         for k in range(n):
88             x[:,i*n*n+j*n+k] = np.array([i*L/n,j*L/n,k*L/n
                   ])
```

Thereby x contains the (initial) position of each particle and i, j and k describe the three dimensional lattice.

We then ran the simulation with $n = 3, 4, 5$ particles per side and the overall number of particles $N = n^3$ for $t = 1.0$ and $dt = 0.01$ and measured the time taken to complete the simulation.

We recorded times of 1.472, 5.328, and 16.704 seconds respectively.

2.5.2 Pure C/C++

Similar to the previous task we were given a template, this time in C++, to modify. In this case we needed to implement the compute_energy() function.

Code block 9: Compute Energy — script: src/ljfluid.cpp

```
73 double compute_energy(double *x, double *v) {
74    double rij[3];
75    double epot = 0, ekin = 0;
76
77    // add up potential energy
78    for (int i = 1; i < N; i++) {
79       for (int j = 0; j < i; j++) {
80          minimum_image(x, i, j, rij);
81          epot += compute_lj_potential(rij);
82       }
83    }
84
85    // add up kinetic energy
86    for (int i = 0; i < N; i++) {
87       ekin += 0.5*(v[i]*v[i]+v[i+N]*v[i+N]+v[i+2*N]*v[i+2*
             N]);
88    }
89
90    return epot + ekin;
91 }
```

We then ran the simulation for $n = 5, 6, 7, 8, 9, 10, 11, 12$ and measured the computation time. For $n = 5$, a time of 0.188 seconds was measured. Almost 10 times faster than python. The data along with a fit of the obvious quadratic trend is included in Fig. 7.

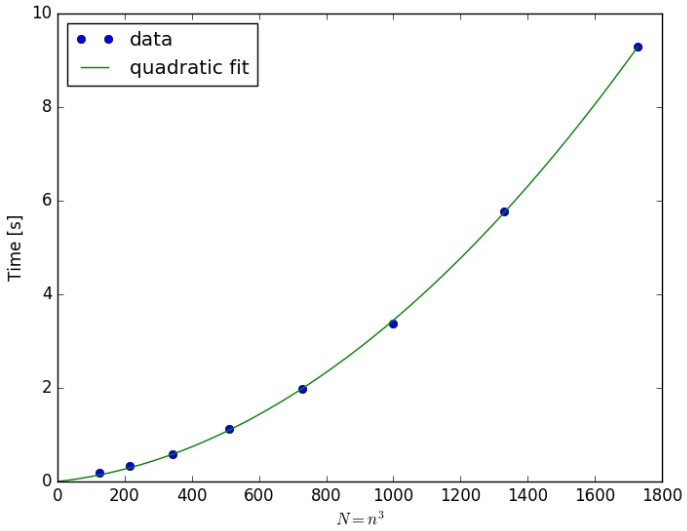

Fig. 7: Plot of computing time for the C++ implementation vs number of particles along with a quadratic fit.

2.6 Cython

Finally we will use cython as an interface between python and C so that the time intensive code is written in C and the control is done with python. We copied the compute_energy() function from the previous task into the supplied template, lines 10 and 21 through 24 were uncommented in `cython1/lj.pyx`, and lines 84 through 87 were uncommented in `cython1/ljfluid.py`. The simulation was run for $n = 5, 10$ and runtimes of 0.568 and 2.724 seconds respectively were recorded. Compared to pure C++ the lower n had a slower runtime while the higher had a quicker runtime.

2.7 Cell Lists and Verlet Lists

Finally, we use cell lists and Verlet lists to further increase the efficiency of our simulation. We had to modify the c_compute_energy() function in order to incorporate the verlet lists.

Code block 10: Compute Energy Verlet script: src/cython2/c_lj.cpp

```
89    double c_compute_energy(double *x, double *v) {
90       double rij[3];
91       double epot = 0, ekin = 0;
92
93       // add up potential energy
94       vector<int>::iterator it = verlet_list.begin();
95       vector<int>::iterator end = verlet_list.end();
96       while (it != end) {
97          int i = *it;
98          ++it;
99          int j = *it;
100         ++it;
101         minimum_image(x, i, j, rij);
102         epot += compute_lj_potential(rij);
103
104
105      // add up kinetic energy
106      for (int i = 0; i < N; i++) {
107         ekin += 0.5*(v[i]*v[i]+v[i+N]*v[i+N]+v[i+2*N]*v[i
               +2*N]);
108      }
109
110      return epot + ekin;
111   }
```

We ran this simulation for $n = 5, 10, 15, 20$ and got runtimes of 0.392, 1.732, 11.916, and 59.960 seconds respectively. This shows a clear advantage in runtimes over the previous methods. All recorded runtimes can be found in `times.txt`.