Molecular Dynamics
FYS-3150

Candidate nr. 18

December 7, 2014

Contents

1 Part 1:

1.1 Task a: Periodic boundary conditions

This task was implemented in the function applyPeriodicBoundaryConditions() and the program can be found at the github page [1].

Since we have limited computational power we are only working on a small piece of argon material, to get away from boundary cases we simulate a system of infinite size by implementing periodic boundary conditions. This is done by confining the atoms in a box and each time a particle reaches the end of the box it is put to the other side.

1.2 Task b: Removing momentum

This task is implemented in the function removeMomentum().

Since all the atoms are given random velocities the net momentum is not zero and the cloud will drift. This is solved by calculating the total momentum of all the atoms, and then subtracting a fraction of the total momentum from each atom so the total momentum ends up being zero.

1.3 Task c: Putting the atoms on a lattice

This task is implemented in the function createFCCLattice()

The noble gas argon should have a stable lattice structure when a solid, by letting the system start in a stable situation we avoid a lot of energy to be infused into the systems temperature due to it minimizing potential energy. The implementation is done by going through several nodes, R_i, put on a grid in the box and then placing 4 atoms around each node. Let c_l be the length between nodes.

$$\mathbf{R}_{ij} = \mathbf{R}_i + \mathbf{r}_j \qquad j = \{1,2,3,4\} \qquad i = \{1,2,...,4\mathrm{N_{atoms}}\}$$

$$\begin{aligned}
\mathbf{r}_1 &= 0\hat{\mathbf{i}} + 0\hat{\mathbf{j}} + 0\hat{\mathbf{k}} \\
\mathbf{r}_2 &= \frac{c_l}{2}\hat{\mathbf{i}} + \frac{c_l}{2}\hat{\mathbf{j}} + 0\hat{\mathbf{k}} \\
\mathbf{r}_3 &= 0\hat{\mathbf{i}} + \frac{c_l}{2}\hat{\mathbf{j}} + \frac{c_l}{2}\hat{\mathbf{k}} \\
\mathbf{r}_4 &= \frac{c_l}{2}\hat{\mathbf{i}} + 0\hat{\mathbf{j}} + \frac{c_l}{2}\hat{\mathbf{k}}
\end{aligned}$$

1.4 Task d: Implementing a Verlet integrator

The Integrator is a widely used integrator in molecular dynamics [3] because of it's properties as a symplectic integrator which means that it conserves areas in phase space very well and it allows stable integration of the equation of motion [4]. Newton's second law for for a particle in our molecule ensemble reads:

$$m\frac{\partial^2 x_i}{\partial t^2} = F_i$$

$$\frac{\partial x_i}{\partial t} = v_i \text{ and } \frac{\partial v_i}{\partial t} = \frac{F_i}{m}$$

The Leapfrog algorithm is a slight continuation on the Verlet algorithm doing the step in two steps. Doing a Taylor expansion around both the step and half the step we get.

$$x_i(t+h) = x_i(t) + hx_i'(t) + \frac{h^2}{2}x_i''(t) + \mathcal{O}\big(h^3\big) \tag{1}$$

$$x_i'(t+\frac{h}{2}) = x_i'(t) + \frac{h}{2}x_i''(t) + \mathcal{O}\big(h^2\big) \tag{2}$$

Inserting equation (2) into (1) to obtain

$$x_i(t+h) = x_i(t) + hx_i'(t+\frac{h}{2}) + \mathcal{O}(h^3) \tag{3}$$

A Taylor first order expansion of the velocity produces the following

$$x_i'(t+\frac{h}{2}) = x_i'(t) + \frac{h}{2}x_i''(t) + \mathcal{O}(h^2) \tag{4}$$

By using equation (4), then (3) and then (4) again we obtain $x_i(t+h)$ and $v_i(t+h)$. The algorithm follows:

$$\begin{aligned}
v_i(t+\frac{h}{2}) &= x_i'(t) + \frac{h}{2}\frac{F_i(t)}{m} + \mathcal{O}(h^2) \\
x_i(t+h) &= x_i(t) + hx_i'(t+\frac{h}{2}) + \mathcal{O}(h^3) \\
v_i(t+h) &= v_i(t+\frac{h}{2}) + \frac{h}{2}\frac{F_i(t+\frac{h}{2})}{m} + \mathcal{O}(h^2)
\end{aligned}$$

1.5 Task e: Computing forces

We will be using the Lennard-Jones potential to approximate the forces between the molecules, which work quite well, given it's simplicity, for neutral particles, especially noble gases, as we are dealing with in this study [2]. The formula is given below.

$$U(r_{ij}) = 4\epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right]$$

At short distances the term to the twelfth power dominates and represents a repulsive force, Paulie exclusion principle, while at longer distances the term to sixth power dominates and represents the attractive van der Waal force. The σ is the distance at which the potential is 0, while ϵ is the depth of the well.

Experimentally the following values for argon has been found: $\begin{cases} \epsilon/k_B & = 119.8\text{K} \\ \sigma & = 3.405\text{Å} \end{cases}$

The force felt between the molecules is given by the negative gradient of the potential.

$$\mathbf{F}(r_{ij}) = -\boldsymbol{\nabla} U(r_{ij})$$

The potential only has a nonzero derivative along $\mathbf{r}_{ij}$, the axis between then particles, so it is natural to evaluate the gradient in that coordinate system before projecting it onto the xyz coordinates used by the program.

$$\begin{aligned}
\mathbf{F}(r_{ij}) &= -4\epsilon\hat{\mathbf{r}}_{ij}\partial_{r_{ij}}\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \\
\mathbf{F}(r_{ij}) &= -4\epsilon\hat{\mathbf{r}}_{ij}\left[\left(-\frac{12}{\sigma}\right)\left(\frac{\sigma}{r_{ij}}\right)^{13} - \left(-\frac{6}{\sigma}\right)\left(\frac{\sigma}{r_{ij}}\right)^{7}\right]
\end{aligned}$$

Then it is projected onto the xyz coordinates, $F_k = \left(F\frac{r_{ij}}{|r_{ij}|}\right)_k = F\frac{k_{ij}}{|r_{ij}|}$, where k_{ij} is the distance between the particles in direction $k = \{x, y, z\}$, and $|r_{ij}|$ is the total distance.

$$F_k = \frac{24}{\sigma}\epsilon\left[2\left(\frac{\sigma}{r_{ij}}\right)^{13} - \left(\frac{\sigma}{r_{ij}}\right)^{7}\right]\frac{k_{ij}}{|r_{ij}|}$$

1.5.1 Algorithm to implement force

The implementation of the force will be along the following steps:

- Calculate $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ for a particle pair
- Calculate $F_{r_{ij}} = \frac{24}{\sigma}\epsilon\left[2\left(\frac{\sigma}{r_{ij}}\right)^{13} - \left(\frac{\sigma}{r_{ij}}\right)^{7}\right]$
- Calculate $F_x = F_{r_{ij}}\frac{k_{ij}}{|r_{ij}|}$ for xyz
- Add the force to both particles force account, halves the necessary computations. One positive one negative

1.6 Task f: Calculating some statistical properties

1.6.1 Kinetic energy

The kinetic energy is found trough adding up the kinetic energy for all the atoms separately. It is straight forward and done in a function, in the statisticsSampler class, called sampleKineticEnergy(). It goes through all the atoms and adds it's contribution to the total kinetic energy.

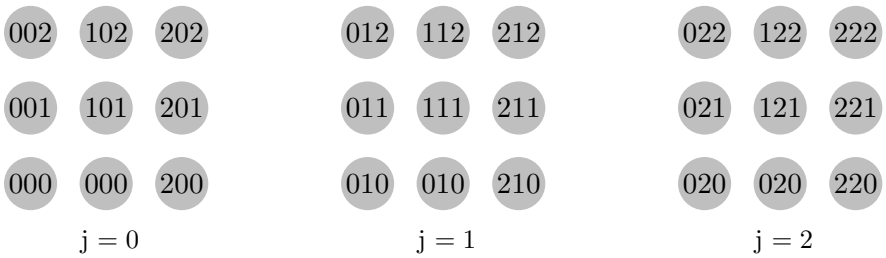
$$E_k = \sum_{i=1}^{N_{atoms}} \frac{1}{2}m_i v_i^2$$

Then it also stores the instantaneous temperature of the substance which is given by the equipartition theorem $\langle E_K\rangle = \frac{2}{3}k_B N_{atoms}T$. By looking at the kinetic energy in a time-instant instead of averaging it we get an instantaneous temperature.

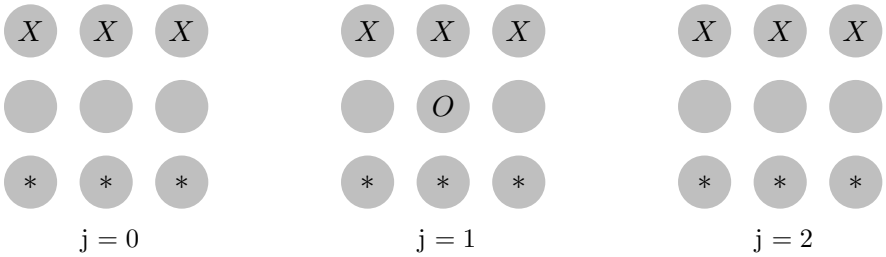
All of this is stored together with the potential energy to a text file "./statisticalResults/statisticalValues.tsv" which can be plotted by the python program plots.py.

1.7 Task g: implementing Neighbor cells

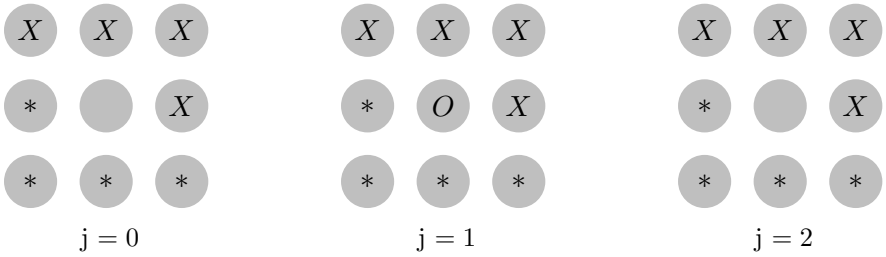
A simple system of 3×3 neighbors, see figure 1a will have all the properties and connections of a larger system, since a neighbor is only connected to the neighbors bordering it. Let us choose the neighbor in the center, 111, and find the neighbors we must go through to find to let it interact with each of it's neighbors once. First we let it interact with the top layer, 1b, and then we note that it is not necessary to let it interact with the bottom layer, since all the cells in that layer will interact with the choosen cell, 111. Then we fill the remainer of the front,



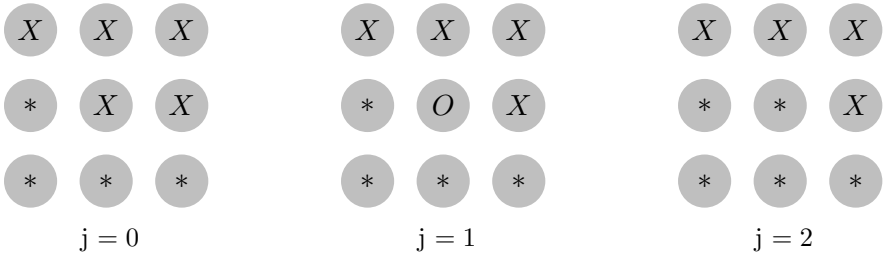
(a) All the neighbors in a 3×3 system divided up into three slices with j constant. The numbers are position in an ijk-grid.



(b) Let the center cell interact with the top layer. 'X' represents a cell that the center cell, 'O' has interacted with. '*' represents a cell that will interact with the center cel, 'O' if it undergoes the same scheme as the center cell 'O'.



(c) Then we let the center cell interact with the rest of the front, i-direction.



(d) Then we let the center cell interact with side, j-direction and all the neighboring cells to cell 111 is interacted with, or interacts with cell 111 when all the cells are run trough.

Figure 1: A schematic explanation of the algorithm to go through all the neighboring cells of a neighbor cell.

N_{atoms}	4	32	108	256	500	864	1372	2048
time (s)	0.003539	0.054251	0.274064	1.22426	3.96325	11.3712	28.3554	62.9434
time (s) with list	-	0.179424	1.74696	2.89621	9.14402	13.3765	33.5662	39.6798

Table 1: This table shows time to compute 100 timesteps with different amount of atoms in the model. The time is increasing much faster than linearly. After the neighborlists are implemented it is slower on very few atoms, but it grows slower and spends less time aabove 2048 atoms.

Table 1 shows the time spent computing with different amount of atoms in the system, as the system increases the time is increasing fast and we can see the need to implement neighbor cells. The time is increasing fast because the program has to calculate the forces between all the atoms and that is of the order $\mathcal{O}(N^2)$. By only calculating the nearby atoms, which will be the ones affecting each other the most, the time taken will mostly increase linearly.

The force is mostly ignorable outside a distance of 3σ, see figure 2.

- Divide box into smaller boxes with $l > 3\sigma$, and put the cell layer class between the system and atoms.

  $$\text{system} \rightarrow \text{cellbox} \rightarrow \text{atoms}$$

- For each box go through all the boxes neighboring boxes. Boxes should be stored in three lists as (i, j, k) and the neighboring boxes to box (i, j, k) are $(i-1, j-1, k-1), (i-1, j-1, k)$ and going through all the combinations to $(i+1, j+1, k+1)$
- Each atom needs an additional box assignment property
- Calculate the forces for all the atoms belonging to the neighboring boxes

1.8 Task g: Implementing a thermostat

After initializing the system, it will after while stabilize with a different temperature than it was initialized in. So we will need to implement a thermostat and let that run for a while to get the system to maintain that temperature that we want to measure statistical properties at. In this project we will implement a Berendsen thermostat, that works by first calculating a scalar value ,γ, for how much the temperature of the system is different from the wanted temperature, see equation (5), where γ is the adjustment scalar, Δt is the timestep, T_Bath and T is the measured and wanted temperatures, τ is a relaxation factor.

$$\gamma = \sqrt{1 - \frac{\Delta t}{\tau}\left(\frac{T_{Bath}}{T} - 1\right)} \tag{5}$$

Then all of the individual atoms velocities are multiplied with the factor γ and the temperature is increased, or decreases, to get closer to the wanted temperature.

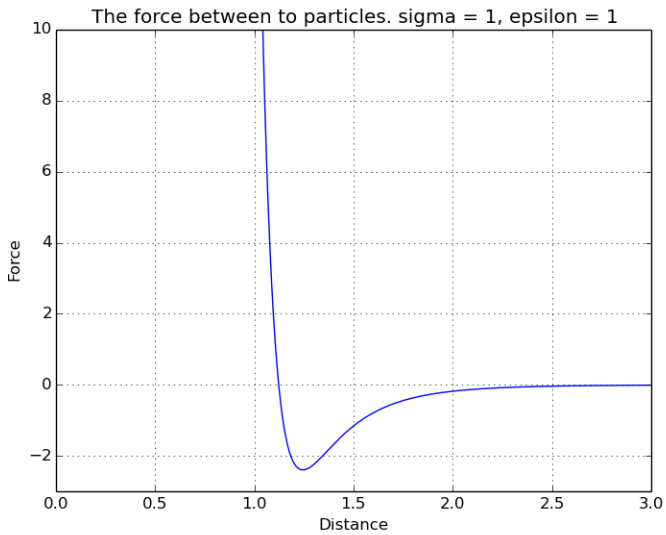


Figure 2: A picture over the force felt between two particles due to the Lennard-Jones potential. At approximately 3 σ of length the magnitude of the force closes to zero and we can ignore the contribution of atoms further away from each other.

2 Part 2: Saving and loading in binary format

2.1 Task a:

The save and load functions, in binary, was implemented as members in the system class and are called load() and save. They load all the positions and velocities of all the particles in the system.

2.2 Task b: Writing the statistical properties to file

In the class statisticsSampler there all the statistical properties are sampled and written to a file. They are stored as "./statisticalResults/statisticalValues.csv"

2.3 Task c: Measure the pressure

The pressure for a molecular system can be calculated with the following equation, (6) (for an NVT ensemble, it also works quite well for an NVE ensemble), where ρ_n is the number density, k_B is the Boltzmann constant, T is the temperature, V is the volume, $\vec{r}_i$ is the position of atom i and $\vec{F}_i$ is the total force on atom i:

$$P = \rho_n k_B T + \frac{1}{3V} \left\langle \sum_{i=1}^{N} \mathbf{F}_i \cdot \mathbf{r}_i \right\rangle \tag{6}$$

The force on one atom is the sum of all the force felt from all the other atoms $\mathbf{F}_i = \sum_{j=1, i\neq j}^{N} \mathbf{F}_{ij}$

$$P = \rho_n k_B T + \frac{1}{3V}\left\langle \sum_{i=1}^{N}\left(\sum_{j=1, i\neq j}^{N} \mathbf{F}_{ij}\right)\cdot \mathbf{r}_i \right\rangle \tag{7}$$

The $\mathbf{r}_i$ can be moved into the $j-$summation since it is not dependent on j

$$P = \rho_n k_B T + \frac{1}{3V}\left\langle \sum_{i=1}^{N}\left(\sum_{j=1, j\neq i}^{N} \mathbf{F}_{ij}\cdot \mathbf{r}_i\right) \right\rangle \tag{8}$$

$$P = \rho_n k_B T + \frac{1}{3V}\left\langle \sum_{i>j} \vec{r}_{ij}\cdot \vec{F}_{ij} \right\rangle, \tag{9}$$

2.4 Task d:

Here we will start up a new system of 4000 atoms with an initial temperature of 300K. Then let it run with no modifications and let it run. As can be seen in figure the kinetic

2.5 Task e: Python framework for running simulations

To be able to easily run several runs of the simulation a small python framework is adapted to do this. It consists of the two python programs run.py and mdconfig.py. The latter consists of a several different configurations to run the c++ simulation of the argon gas. It has a MD class which contains configurations to tell the simulation to create a new FCC lattice, run the simulation with the Berendsen thermometer on and one for letting it run, along with saving and loading functions. run.py is the overarching program where what we want to simulate is defined and the functions from mdconfig.py is called.

2.6 Task f: Measure the heat capacity

2.7 Task g: Pressure vs number density

As the lattice constant is changed, the volume of our gas will change and subsequently the number density. The length in one direction is dependent on the number of unit cells and the distance between them. Let V, b and U be the volume, lattice constant and unit cells respectively.

$$V = (bU)^3 \tag{10}$$

The number density is then

$$\rho_n = \frac{N_a}{V} = \frac{N}{U^3} b^{-3} \tag{11}$$

The number of atoms is dependent on the number of unit cells chosen, with four per unit cell in each dimension

$$\rho_n = \frac{4 * U^3}{U^3} b^{-3} = \frac{4}{b^3} \tag{12}$$

A Unit scheme

The program uses a more natural set of units which let's Boltzmann's constant be 1.

$$1 \text{ mass unit } = 1 \text{ a.m.u } = 1.661 \times 10^{-27} \text{ kg} \tag{13}$$

$$1 \text{ length unit } = 1.0 = 1.0 \times 10^{-10} \text{ m} \tag{14}$$

$$1 \text{ energy unit } = 1.651 \times 10^{-21} \text{ J} \tag{15}$$

$$1 \text{ temperature unit } = 119.735 \text{ K} \tag{16}$$

All the other units are then expressed in term of these units.

Boltzmann constant translates between temperature and energy, in SI-units it is $k_B = 1.381 \times 10^{-23}$ J K^{-1}, which in the previously defined units becomes

$$k_B = 1.381 \times 10^{-23} \left(\text{ J K}^{-1} \right) \left(\frac{\text{energy unit}}{1.651 \times 10^{-21} \text{ J}} \right) \left(\frac{119.735 \text{ K}}{\text{temperature unit}} \right) \tag{17}$$

$$k_B = 1 \, \frac{\text{energy unit}}{\text{temperature unit}} \tag{18}$$

References

[1] Gulliks fys3150 github page
`https://github.com/Gullik/molecular-dynamics-fys3150`.

[2] Wikipedia `http://en.wikipedia.org/wiki/Lennard-Jones_potential`.

[3] Morten Hjorth-Jensen. Computational physics lecture notes fall 2014.

[4] Dominik Marx and Jurg Hutter. Ab initio molecular dynamics: Theory and implementation, 2009.