Molecular Dynamics Project

FYS-MEK1110 — Mechanics

Tentative deadline: 15th March 2020
Last updated: 17th January 2020

Part 0 Practical information

This project is meant as a more challenging alternative to the first three mandatory assignments for those who feel that they already have a good grasp of high school physics, basic scientific programming and the Forward Euler method. The workload is (hopefully) approximately equal to the combined workload of the first three assignments.

A special weekly group session[1] will focus on this project, where you can discuss with fellow students and get help from teaching assistants. Additionally, you are encouraged to use the Piazza/Padlet platforms for asking your own questions and answering the questions of other students.

If you at some point find that this project is too difficult or time-consuming for you, the work you have already done may replace one or two normal assignments depending on how far you have come. Let us know, and we will find a solution.

Since this project dives deeply into one subject, it touches on fewer topics from the syllabus than the ordinary assignments. If you choose to do this project, it is therefore especially important that you also do the weekly exercises and attend normal group sessions.

Python and Matlab are the natural choices of programming language for this project. You are allowed to choose any other language, especially fast languages such as Fortran and C++, but you cannot expect all teaching assistants to know all programming languages.

When writing your programs, you should keep two things in mind:

- Reusability: This project contains many exercises, but most of them are variants of or build upon earlier exercises, so you will save a lot of time if your implementation is easy to extend or adapt to a new problem.
- Efficiency: The two last parts of the project require larger simulations which will run for hours (or even days) if your program is slow. Try therefore to think of efficiency, and use vectorised operations with Numpy arrays whenever possible. A small note on this can be found here. Use your own time wisely as well, e.g. by writing answers to the exercises while your programs are running.

You are free to choose the format of your report - either in the form of a scientific report (inspiration can be found from page 4 here), or answer each exercise separately. For the latter, a good option is to use Jupyter Notebook[2], where you can run code and show results directly in your document. Otherwise, your code should be added to your delivery separately.

In your report you are expected to give thorough descriptions of both the theory and methods you have used, as well as critical discussions of your results.

In the end, we hope that you will enjoy the project and your introduction to the world of molecular dynamics.

Good luck!

[1] https://www.uio.no/studier/emner/matnat/fys/FYS-MEK1110/v20/timeplan/index.html
[2] https://jupyter.org/

Part 1 Introduction

In this project, you will learn the basics of a simulation technique called molecular dynamics (MD). Molecular dynamics is a method actively used in research here at the Department of Physics, yet its basic principle can be understood and implemented with the background of a first-year physics student.

Molecular dynamics is based on the assumption that even atoms move according to the laws of Newton, given the correct model for interactions. The goal of this project is to model an argon gas, where the atoms interact according to the famous Lennard-Jones potential,

$$U(r) = 4\varepsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right), \tag{1}$$

where r is the distance between two atoms, $r = \left\|\vec{r}_i - \vec{r}_j\right\|$. σ and ε are parameters which determine which chemical compound is modelled. This potential is a good approximation for noble gases.

1a) Understanding the potential

i. Plot the potential as a function of r with $\varepsilon = 1$ and $\sigma = 1$, for example for $r \in [0.9, 3]$.

ii. The behaviour of $U(r)$ is vastly different for $r < \sigma$ and $r > \sigma$. Which term in the potential, equation (1), dominates in each case and what is the effect?

iii. Find and characterise the equilibrium points of the potential.

iv. Describe qualitatively the motion of two atoms which start at rest separated by a distance of 1.5σ. What if they start with a separation of 0.95σ? (Hint: use the graph of the potential.)

v. Describe the shape of the potential close to the stable equilibrium point. Can you think of other force(s) with the same behaviour?

1b) Forces and equations of motion

i. Find the force on atom i at position $\vec{r}_i$ from atom j at position $\vec{r}_j$.

ii. Show that the equation of motion for atom i is

$$\frac{\mathrm{d}^2\vec{r}_i}{\mathrm{d}t^2} = \frac{24\varepsilon}{m}\sum_{j\neq i}\left(2\left(\frac{\sigma}{\left\|\vec{r}_i - \vec{r}_j\right\|}\right)^{12} - \left(\frac{\sigma}{\left\|\vec{r}_i - \vec{r}_j\right\|}\right)^{6}\right)\frac{\vec{r}_i - \vec{r}_j}{\left\|\vec{r}_i - \vec{r}_j\right\|^2}. \tag{2}$$

1c) Units

As you may remember from MAT-INF1100, numerical accuracy is reduced when computing with values which are many orders of magnitude apart. This is often an issue in physics, and molecular

dynamics is no exception. For example, the mass of argon is smaller than 10^{-25} kg, while typical length scales are on the order of nanometres, 10^{-9} m.

The remedy is to change units so that most quantities are close to 1. From equation (1) on the previous page it is clear that σ and ε are the typical scales for length and energy.

i. Introduce the scaled coordinates $\vec{r}_i\,' = \vec{r}_i/\sigma$ and show that the equation of motion can be rewritten in terms of these coordinates as

$$\frac{\mathrm{d}^2\vec{r}_i\,'}{\mathrm{d}t'^2} = 24\sum_{j\neq i}\left(2\left\|\vec{r}_i\,' - \vec{r}_j\,'\right\|^{-12} - \left\|\vec{r}_i\,' - \vec{r}_j\,'\right\|^{-6}\right)\frac{\vec{r}_i\,' - \vec{r}_j\,'}{\left\|\vec{r}_i\,' - \vec{r}_j\,'\right\|^2}, \tag{3}$$

where $t' = t/\tau$ for a suitable choice of τ.

ii. What is the characteristic time scale τ, and what is its value for argon, which has $\sigma = 3.405$ Å (1 Å $= 1\cdot 10^{-10}$ m), $m = 39.95$ u (1 u $= 1.66\cdot 10^{-27}$ kg) and $\varepsilon = 1.0318\cdot 10^{-2}$ eV (1 eV $= 1.602\cdot 10^{-19}$ J)?

Part 2 Two-atom simulations

2a) Implementation

i. Write a function which solves equation (3) for two atoms and finds the positions and velocities of the atoms as a function of time. Implement three different integration methods: Euler, Euler-Cromer and Velocity-Verlet (see appendix C on page 11 for a description of the latter).

2b) Motion

i. Simulate the motion of two atoms which start at rest separated by a distance of 1.5σ. Use $\Delta t' = 0.01$, simulate until $t' = 5$ and integrate with the Euler-Cromer method.

ii. Plot the distance between the atoms as a function of time.

iii. How does the motion fit with your expectations from exercise 1a) on the preceding page?

iv. Repeat the previous tasks, but now with an initial separation of 0.95σ. Explain your results.

2c) Energy

i. Plot the kinetic, potential and total energy as a function of time for the two cases in the previous section.

ii. Theoretically speaking, should the total energy be conserved? Why, or why not? What about momentum?

iii. Does your program fulfil this? If not, what could be the cause?

iv. Simulate the same system as in exercise 2b) on the previous page with the Euler, Euler-Cromer and Velocity Verlet algorithms, and compare graphs of the total energy as a function of time.

v. Find the largest time step that keeps stable motion and conserves energy for all three methods (small fluctuations in energy are allowed as long as they are periodic and don't increase/decrease with time). Discuss your results.

vi. Link your experimentation to a brief discussion of the pros and cons of the three methods, both physically and computationally.

The Velocity-Verlet method should be used for the rest of the project.

2d) Visualisation

i. Extend your implementation such that it writes to an `xyz`-file at each timestep (see appendix A on page 11).

ii. Visualise the results of your simulations using Ovito (see appendix B on page 11).

Part 3 Large systems

3a) Implementation

i. Implement a solver of equation (3) on the preceding page for N atoms, given initial positions and velocities.

ii. Use Newton's third law to reduce the number of force calculations.

As you will experience, it takes a lot longer to simulate N atoms than two — in fact the time increases as N^2. One very simple way to reduce simulation times is to look at the expression (or plot) of $U(r)$ and see that it goes very rapidly towards zero as r increases. This means that atoms far apart interact weakly, and the forces between them can be ignored.

iii. Extend your implementation such that atoms more than 3σ apart do not interact.

This effectively sets the potential energy $U(r)$ to be 0 for $r \geq 3\sigma$. Since $U(3\sigma)$ is not exactly equal to 0, the potential energy becomes discontinuous, which breaks energy conservation. The solution is to use a shifted potential, i.e. adding a constant such that $U(3\sigma)$ is exactly zero.

iv. Plot the shifted potential and the corresponding force to verify your implementation of the cut-off.

v. Does the shift of the potential described above impact the force calculations?

3b) Verification

i. Reproduce your results for the 2-atom model from the previous section to verify your implementation.

ii. Simulate the motion of four atoms starting at rest from the positions $[1, 0, 0]$, $[0, 1, 0]$, $[-1, 0, 0]$ and $[0, -1, 0]$.

iii. Visualise the results in Ovito, describe and explain the motion.

iv. Plot the potential, kinetic and total energy as a function of time, and comment on the energy conservation.

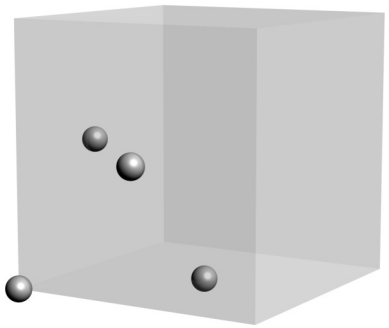
v. Repeat the above exercises with a small perturbation in the initial positions, such that the first atom starts at $[1, 0.1, 0]$.

3c) Initialisation

While we are interested in simulating liquid argon, which will not be in an ordered structure, the simplest choice of initial positions is a regular crystal structure. Our choice of structure is the face-centred cubic lattice, as this is the crystal structure of solid argon.

The smallest repeating unit is called a unit cell, and each unit cell contains four atoms. When creating a structure of $n \times n \times n$ unit cells, the atoms should be placed at

$$\begin{bmatrix} i & j & k \\ i & 0.5+j & 0.5+k \\ 0.5+i & j & 0.5+k \\ 0.5+i & 0.5+j & k \end{bmatrix} \cdot d$$



where i, j and k run from 0 to $n-1$ and d is the size of one unit cell. One atom is placed at the bottom left corner of the unit cell, and the other three at the centre of the three connected walls (see drawing above). This structure will contain $4n^3$ atoms in total, and the total size of the simulation box will be $(nd)^3 = L^3$.

i. Write a function which takes n and L (or n and d) as arguments and returns the positions of $4n^3$ atoms on a face-centred cubic lattice.

ii. Verify your implementation by calling your function for $n = 3$ and $L = 20$, writing the resulting positions to an `xyz`-file and looking at the result in Ovito. Your system should contain $4 \cdot 3^3 = 108$ atoms.

iii. Show that the unit cell size corresponding to the density $\rho = 1.374$ g/cm^3 is $d = 1.7\sigma$. This will be used in the remaining parts of the project.

3d) Many atoms, open boundary

i. Simulate 256 atoms starting from rest, and visualise the results.

ii. Plot the potential, kinetic and total energy as a function of time. What is the main difference from the energy graphs for two and four atoms?

3e) Boundary conditions

In the previous exercise, you probably observed that the atoms immediately spread out into a large volume, possibly with some continuing to infinity (and beyond). This is not the behaviour we want. Ideally, we would like to model the bulk behaviour of argon by the use of periodic boundary conditions, although it can be finicky to implement this efficiently.

To simulate bulk behaviour properly, the atoms should not only be constrained to move inside the box, but should also feel forces from neighboring images (copies) of the same box. i.e. if an atom is close to the right-facing wall, it should feel forces from atoms close to the left-facing wall, as if the two walls were connected. To efficiently calculate the shortest distance between atoms i and j across boundaries, the following trick should be used

```
dx = x[j] - x[i]
dx = dx - round(dx/L)*L
```

where `L` is the size of the box, and `round` rounds up to the nearest integer (see if you understand the logic behind this calculation!). Periodic boundary conditions also mean that if an atom leave the box on one side, it should reenter on the opppsite side.

If the implementation of periodic bounadry conditions proves to be too diffucult or too slow, it's also possible to use *reflective* boundary conditions. Although it doesn't simulate bulk behaviour the same way, it still gives reasonable results for the quantities that we want to measure for larger systems.

Having reflective boundary conditions simply means that atoms that would otherwise leave the box, are turned around. Calculations of forces across boundaries are also neglected in this case.

i. Implement either periodic or reflective boundary conditions (or both) in your program.

ii. Run a simulation with 108 atoms and verify visually that your implementation works. Give the atoms some initial velocities of your own choosing.

Part 4 Science

By now, you will have a well-functioning (if not terribly efficient) set of tools for running molecular dynamics simulations. It is now time to apply these tools to real problems. The goal is to reproduce some of the main results from the first article containing proper molecular dynamics ("landmark simulations" according to Wikipedia[3]), written by A. Rahman in 1964[1].

[3] `https://en.wikipedia.org/wiki/Molecular_dynamics#History`

Section 4b) and 4c) (marked with *) describes two different ways to find the *diffusion constant*. You may choose to do only one of the them, but we'll encourage you to do both. It's worth noting however that a functioning implementation of periodic boundary conditions is needed for section 4c.

4a) Temperature

Measurement: Temperature is one of the most important concepts in thermodynamics, and you will learn much more about it in FYS2160. As you may have learnt already, temperature measures the vibrations of atoms. This is formalised through the equipartition theorem, which for a monoatomic gas such as argon states that

$$\langle K \rangle = \tfrac{3}{2} k_\mathrm{B} T,$$

where $\langle K \rangle$ is the average kinetic energy, T is the temperature and k_B is Boltzmann's constant. In a molecular dynamics simulation, the temperature can be calculated by using the equipartition theorem "backwards", i.e.

$$T = \frac{2 \langle K \rangle}{3k_\mathrm{B}} = \frac{\langle mv^2 \rangle}{3k_\mathrm{B}} = \frac{m}{3k_\mathrm{B}N} \sum_i v_i^2.$$

The sum runs over all atoms.

Units: Note that since we are using reduced coordinates such as $\vec{r}\,' = \vec{r}/\sigma$ and $t' = t/\tau$, a direct calculation from the equation above gives a reduced temperature $T' = T/T_0$, where $T_0 = \varepsilon/k_\mathrm{B} = 119.7$ K. In reduced units, the temperature expression is simplified to

$$T' = \frac{1}{3N} \sum_i v_i'^2.$$

i. Extend your implementation with the calculation of temperature.

Initialisation of velocities: Velocities in a gas are usually distributed according to a normal distribution, i.e. a Gaussian function, as you can see in figure 1 of [1]. In order to initialise a system with a given temperature, each component of the velocity of each atom should be randomly chosen from a normal distribution with mean zero and standard deviation $\sqrt{kT/m}$. The following `numpy` command achieves this in reduced units:

```
v0 = np.random.normal(0, sqrt(T), size=(N,3))
```

ii. Run a simulation with 108 atoms and an initial temperature of 300 K. Plot the temperature as a function of time.

iii. Find an initial temperature that makes the equilibrium temperature approximately equal to the 94.4 K used in [1]. Plot the temperature as a function of time.

4b) *Velocity autocorrelation and diffusion coefficient

The velocity autocorrelation, denoted $A(t)$, is a measure of how similar the distribution of velocities is to the initial distribution. It is shown in figure 4 of [1]. $A(t=0) = 1$, since the velocity distribution at $t = 0$ *is* the initial distribution, while $A(t)$ decreases rapidly for $t > 0$ as the atoms collide with each other and the velocities become less and less similar to the initial distribution.

Mathematically, the velocity autocorrelation is defined as

$$A(t) = \left\langle \frac{\vec{v}(t) \cdot \vec{v}(0)}{\vec{v}(0) \cdot \vec{v}(0)} \right\rangle = \frac{1}{N} \sum_i \frac{\vec{v}_i(t) \cdot \vec{v}_i(0)}{\vec{v}_i(0) \cdot \vec{v}_i(0)}, \tag{4}$$

where the sum runs over all atoms.

i. Add the calculation of the velocity autocorrelation to your implementation.

ii. Run a simulation with e.g. 256 atoms (more if you can), and plot the velocity autocorrelation as a function of time. Compare with figure 4 of [1].

One cause of deviations is that the initial configuration of positions and velocities ($\vec{v}_i(0)$ in equation (4)) should be an equilibrium distribution.

iii. Use the final positions and velocities from the previous simulation as initial positions and velocities for a new simulation, and calculate and plot the new velocity autocorrelation.

iv. If your plot is very noisy (and your program does not run for too long), redo the two above simulations multiple times and average the velocity autocorrelation.

The article also contains calculations of the diffusion coefficient D, as shown in figure 3. Here, it is calculated as the slope of the mean squared displacement as a function of time. This requires periodic boundary conditions, which we have chosen to omit from this project. Fortunately, the diffusion coefficient can also be obtained from the velocity autocorrelation, via the Green-Kubo relation,

$$D = \frac{1}{3} \int_0^\infty A(t) \, \mathrm{d}t \, .$$

While it is not possible to integrate to ∞ in a molecular dynamics simulation, the rapid decrease of $|A(t)|$ ensures that a finite integral will give a good approximation. Use your previous plot of $A(t)$ to determine a reasonable upper bound.

v. Estimate the diffusion coefficient from your previously calculated velocity autocorrelation. Compare with the result from [1].

4c) *Mean squared displacement and diffusion coefficient

A more intuituve and direct way to calculate the diffusion in the system is via the *mean squared displacement*, defined as

$$\langle r^2(t)\rangle = \langle(\vec{r}(t) - \vec{r}(t_0))^2\rangle = \frac{1}{N}\sum_{n=1}^{N}(\vec{r}_n(t) - \vec{r}_n(t_0)))^2. \tag{5}$$

This measure tells us the average distance the atoms has traveled after a time t as compared to their positions at a reference time, t_0. It's important that t_0 is set at a time where the system is already at equilibrium.

Important note: Periodic boundary conditions are required to calculate the msd, as we need to keep track of where, and how many times the atoms has crossed the boundaries. This means counters for all atoms in all three directions are needed.

i. Add the calculation of the mean squared displacement to your implementation.

Through the theory of brownian motion, it can be shown that the mean squared displement is linked to the diffusion constant D by the following relation (in three dimensions):

$$\langle r^2(t)\rangle = 6Dt \quad \text{when } t \to \infty. \tag{6}$$

As with the velocity autocorrelation, the relation to the diffusion constant implies a simulation of infinite time, but the rapid convergence of the msd ensures a good approximation also in this case.

ii. Implement the calculation of the diffusion constant, and run a simulation for 864 atoms (fewer is also fine if the runtime is slow). How does your result compare to the one in [1]?

iii. (Optional) If your program isn't too slow, calculate the diffusion constant as a function of temperature $D(T)$ for equilibrium temperatures in the range of $T = [50K, 120K]$. Plot the result and describe what you see. Google the element of argon, and see if you find something that's supposed to happen in the given temperature range. Can you link this to your results?

4d) Radial distribution function

The radial distribution function $g(r)$ shown in figure 2 of [1] describes the distribution of distances between an atom and its neighbours. It is defined as the ratio of the density at a distance r from an atom and the average density, i.e.

$$g(r) = \frac{V}{N}\frac{n(r)}{4\pi r^2 \Delta r}, \tag{7}$$

where V is the total volume, N the total number of particles and $n(r)$ the average number of particles at a distance between r and $r + \Delta r$. $g(r)$ should be calculated for all atoms and then averaged. An example implementation is given below. The output should be averaged over many timesteps for a smooth result. Note that while $g(r)$ should approach 1 when $r \to \infty$ in bulk, the finite system size causes $g(r)$ to decay for large r.

i. Run a simulation with as many atoms for as long as you can. Calculate the radial distribution function $g(r)$, plot the result and compare with figure 2 of [1].

```
def rdf(bin_edges, r, V):
    """
    bin_edges = edges of bins. Typically np.linspace(0, rc, num_bins+1)
                for some cut-off rc.
    r = Nx3-array of positions of atoms at a given timestep.
    V = volume of system.
    """

    N = r.shape[0]

    bin_centres = 0.5 * (bin_edges[1:] + bin_edges[:-1])
    bin_sizes = bin_edges[1:] - bin_edges[:-1]

    n = np.zeros_like(bin_sizes)

    for i in range(N):
        dr = np.linalg.norm(r - r[i], axis=1)     # Distances from atom i.
        n += np.histogram(dr, bins=bin_edges)[0]  # Count atoms within each
                                                  # distance interval.

    n[0] = 0

    # Equation (7) on the preceding page:
    rdf = V / N**2 * n / (4 * np.pi * bin_centres**2 * bin_sizes)

    return rdf
```

Appendix

A Data file format

The `xyz`-format is a semi-standard format for storing data from molecular dynamics simulations. Each time step is stored in the following format, and there are no blank lines between timesteps:

- A line containing the number of atoms (an integer).
- An ignored line (this line is usually written as a header for the subsequent columns).
- One line for each atom, containing the atom type and the x-, y- and z-coordinates.

For two atoms simulated over three timesteps, where `xij` represents the x-coordinate of atom j at timestep i, the file would look like this:

```
2
type  x   y   z     <--- This line is a read as a comment, and therefore ignored.
Ar   x11 y11 z11
Ar   x12 y12 z12
2
type  x   y   z
Ar   x21 y21 z21
Ar   x22 y22 z22
2
type  x   y   z
Ar   x31 y31 z31
Ar   x32 y32 z32
```

B Visualisation

Files written in the `xyz`-format can be read using the Ovito visualisation tool. It can be downloaded and installed from `https://ovito.org/index.php/download`.

When the installation has finished, simply open Ovito, click "File" $\rightarrow$ "Load File" and choose your `xyz`-file. Edit the column mapping in the dialogue if necessary. When the atoms have appeared on your screen, check the box named "File contains time series" on the right-hand side, press $\triangleright$ and watch your atoms move around!

C Velocity-Verlet

The Velocity-Verlet integration scheme is based on a second-order Taylor polynomial. With $\vec{r}_i(t)$ denoting the position of atom i at a time t, the second-order Taylor expansions of position and velocity can be written as

$$\vec{r}_i(t + \Delta t) \approx \vec{r}_i(t) + \vec{v}_i(t)\Delta t + \tfrac{1}{2}\vec{a}_i(t)\Delta t^2$$

$$\vec{v}_i(t+\Delta t) \approx \vec{v}_i(t) + \vec{a}_i(t)\Delta t + \tfrac{1}{2}\vec{a}_i'(t)\Delta t^2.$$

There is no explicit expression for $\vec{a}'(t)$. It can, however, be approximated using our old friend

$$\vec{a}\,'(t) \approx \frac{\vec{a}(t+\Delta t) - \vec{a}(t)}{\Delta t}.$$

Since the acceleration is independent of the velocity, the newly updated position, $\vec{r}(t+\Delta t)$, is sufficient to calculate $\vec{a}(t+h)$. Inserting this into the expression for $\vec{v}(t+\Delta t)$, we get

$$\begin{aligned}\vec{v}_i(t+\Delta t) &\approx \vec{v}_i(t) + \vec{a}_i(t)\Delta t + \tfrac{1}{2}(\vec{a}_i(t+\Delta t) - \vec{a}_i(t))\Delta t \\ &= \vec{v}_i(t) + \tfrac{1}{2}(\vec{a}_i(t) + \vec{a}_i(t+\Delta t))\Delta t.\end{aligned}$$

The discretised algorithm then becomes

$$\begin{aligned}\vec{r}_{i,j+1} &\approx \vec{r}_{i,j} + \vec{v}_{i,j}\Delta t + \tfrac{1}{2}\vec{a}_{i,j}\Delta t^2 \\ \vec{v}_{i,j+1} &\approx \vec{v}_{i,j} + \tfrac{1}{2}\left(\vec{a}_{i,j} + \vec{a}_{i,j+1}\right)\Delta t,\end{aligned}$$

where $\vec{r}_{i,j}$ is the position of atom i at timestep j. In your implementation, you should avoid having to calculate the acceleration more than once per timestep.

References

[1] Aneesur Rahman. "Correlations in the Motion of Atoms in Liquid Argon". In: *Physical Review* 136.2A (Oct. 1964), A405–A411. DOI: 10.1103/physrev.136.a405. URL: https://doi.org/10.1103%2Fphysrev.136.a405.