Molecular dynamics simulation of Argon

Python

AP3082 Computational Physics

by

Abi Kanagaratnam & Jim Koning

Team of Applied Physics (MSc)

at the Delft University of Technology.

Group Members: Kanagaratnam, Abi (4481968)
Koning, Jim (4481674)

Assignment: March 21st, 2021

TU Delft

Abstract

The goal of this computational assignment is to find the diffusion coefficient using a working molecular dynamics simulation of Argon. This will be determined using the mean-square displacement of Argon atoms in a FCC lattice configuration at a temperature of 1K. The mean-square displacement over time will give us a good indication for what phase (solid, liquid or gas) the Argon atoms are in. We have found that the Argon atoms behave as solid particles at a temperature of 1K, with a diffusion coefficient of ....

Contents

Introduction

In this part of the report certain elements will be explained in order to understand the rest of the report.

Choice of particles

Our aim is to choose particles that are easily and simply defined for classical mechanics simulations. For this assignment, the choice has been to let our particles be atoms. Molecules would have been an option as well. However, molecules would be subject to complexities such as asymmetric shape and molecular-level bonds (such as hydrogen bonds). To avoid such complexities, we want to avoid using molecules and atoms that could form into molecules. Atoms that are unlikely to form into molecules are noble gases. Therefore, our choice of which particle to choose is now limited to an atom which is a noble gas as well. Historically, one of the most studied noble gases is Argon. This would help us to compare whether our simulation meets the expectations from prior studies. For verification and simplicity reasons, our simulation will be using Argon atoms.

Chapter 1

Methodology

Theory

In this section the theory for running our molecular dynamics simulation of Argon atoms will be explained. Also, the theory for the mean-square displacement and the diffusion will be given.

1.1 Interaction of Argon atoms

Argon atoms are neutral in charge, therefore there is no Coulomb interaction present. However, Argon atoms do have a dipole-moment caused by the electrons and the nucleus. This gives them an attractive force. Though, Argon atoms also have a repulsive force which prevents Argon atoms to get too close to each other and consequently prevents them from forming molecules. The attractive potential has a proportionality to the distance of two atoms given as:

$$U(r) \propto \frac{1}{r^6}, \tag{1.1}$$

where r is the distance between the two atoms.

1.1.1 Lennard-Jones potential

Combining the attractive force and the repulsive force of Argon atoms will result in the following potential.

$$U(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right], \tag{1.2}$$

where $\epsilon/k_B = 119.8$K and $\sigma = 3.405$ Angstrom, with $k_B = 1.38064852 \cdot 10^{-23}$. More on these values can be found in the chapter **nummersss**
This potential is also known as the Lennard-Jones potential

1.1.2 Infinite space system

In our simulation we would want to be able to simulate an infinite amount of space where our Argon atoms could be in. However, infinite is an unknown quantity to computer. Therefore, we simplify our simulations by implementing periodicity. With the periodic boundary condition in place, the Argon atoms in our main simulation box will also be affected by the Argon atoms in the surrounding simulation boxes (due to periodicity). In the end, it will appear as if there is copies of our main simulation box around the main simulation box.

1.1.3 Initial locations of atoms

The initial locations of the atoms can be quite important. If the atoms start with a location very close to an other atom, the velocities could increase to extremely high values in the first few timesteps. Due

to the step iterative method of calculating locations and velocities, a particle with a high velocity can 'teleport' to a location far away from its previous location. The simulation is only valid, if the potential change the particles 'feel' is gradual to conserve energy. More on the step iterative method in the chapter **implementation algorithms.**

For starting locations, a FCC lattice is chosen. This is a physical representation of Argon, and approximately, depending on the lattice constant, resembles the locations where the atoms are in a equilibrium state.**source needed**

1.2 Mechanics

We will be using Newton's laws to simulate the classical motion of Argon particles. This will be explained next.

1.2.1 Newton's equation

The motion of the particles is given as follows:

$$m\frac{d^2\mathbf{x}_i}{dt^2} = \mathbf{F}(\mathbf{x}_i) = -\nabla U(\mathbf{x}_i), \tag{1.3}$$

where i is the index of a particle and $\mathbf{F}$ is the sum of the forces acting on particle i.
We see that the potential is dependant on the distance r. In a three dimensional situation we can see that $r = \sqrt{x^2 + y^2 + z^2}$ so the following relation holds:

$$\nabla U(r) = \frac{dU}{dr}\frac{\mathbf{x}}{r}, \tag{1.4}$$

1.2.2 Implementation algorithms

To implement the previously mentioned Newton's equation into our simulation we need to use an algorithm. This will be explained next.

1.2.2.1 Euler's method

The Euler's algorithm is given using the following equation:

$$t_{n+1} = t_n + h, \tag{1.5}$$

which results in the following two equations:

$$\mathbf{x}_i(t_{n+1}) = \mathbf{x}_i(t_n) + \mathbf{v}_i(t_n)h \tag{1.6}$$

and

$$\mathbf{v}_i(t_{n+1}) = \mathbf{v}_i(t_n) + \frac{1}{m}\mathbf{F}(\mathbf{x}_i(t_n))h, \tag{1.7}$$

where h is the time between time-steps and n is the index of the time-step. $\mathbf{x}_i$ and $\mathbf{v}_i$ are the position vector and the velocity vector, respectively.

1.2.2.2 Velocity-Verlet algorithm

To improve the accuracy of the Euler's method, the Verlet algorithm will be used. The differences in accuracy are minor in comparison. However, the Verlet algorithm uses so-called symplectic integrators that works well with time-evolving an Hamiltonian system. The classical mechanics we will use in our simulation is an example of such Hamiltonian system. When slightly modifying the Verlet algorithm, we will obtain the velocity-Verlet algorithm. This is given as follows:

$$\mathbf{x}_i(t_{n+1}) = \mathbf{x}_i(t_n) + h\mathbf{v}_i(t_n) + \frac{h^2}{2}\mathbf{F}(\mathbf{x}_i(t_n)) \tag{1.8}$$

and

$$\mathbf{v}_i(t_{n+1}) = \mathbf{v}_i(t_n) + \frac{h}{2}(\mathbf{F}(\mathbf{x}_i(t_{n+1})) + \mathbf{F}(\mathbf{x}_i(t_n))) \tag{1.9}$$

Note that these equations are given in dimensionless units. Dimensionless units will be explained next.

1.3 Dimensionless units

Since our simulations run with an atomic scale system, the numbers that are used for our calculates will also have very different orders of magnitudes. Computers do not seem to work well with numbers that have higher orders of magnitudes; it could lead to floating-point errors or round-off errors. Also working on a program with numbers that have higher orders of magnitudes is more prone to human errors as well. Therefore, we will be using ”reduced” units, also called: dimensionless units.
We will be expressing time in $(\frac{m\sigma^2}{\epsilon})^{1/2}$, length in σ and energy in ϵ, these will have the following values:

$$t_{\text{DIMLESS}} = (\frac{m\sigma^2}{\epsilon})^{1/2} = 2.15 \cdot 10^{-12}\text{s}, \tag{1.10}$$

$$E_{\text{DIMLESS}} = \epsilon = 1.65 \cdot 10^{-21}\frac{\text{J}}{\text{K}}, \tag{1.11}$$

and

$$l_{\text{DIMLESS}} = \sigma = 3.405 \cdot 10^{-10}\text{m}. \tag{1.12}$$

From these the dimensionless unit for velocity can be determined as well as follows:

$$v_{\text{DIMLESS}} = \frac{l_{\text{DIMLESS}}}{t_{\text{DIMLESS}}} = \frac{\sigma}{(\frac{m\sigma^2}{\epsilon})^{1/2}} = \sqrt{\frac{\epsilon}{m}}. \tag{1.13}$$

1.4 Simulation constants

The objective of this paper was to reproduce some observables from the book *Computational Physics* published by *Jos Thijssen*. Thus, based on this, some of the following simulation constants were introduced.

1.5 Temperature rescaling

In order to allow our simulation to have phase transition for the Argon particles, it should be possible to allow the system to reach a desired target kinetic energy using a target temperature. Since the kinetic energy is a function of the temperature but also a function of velocity, it will mean that a change in temperature will result in a change in kinetic energy and consequently a change in velocity as well. The kinetic energy is given as:

$$E_{\text{kin}} = (N-1)\frac{3}{2}k_B T, \tag{1.14}$$

where N is the amount of particles. The kinetic energy is also given as:

$$E_{\text{kin}} = \sum_i \frac{1}{2}mv_i^2, \tag{1.15}$$

where v_i is the velocity of the individual particles.

Since the kinetic energy is dependent on the temperature, we can define a target temperature using a self-defined temperature. To make the velocities scale towards the target kinetic energy, we need to define a scaling factor as follows:

$$\lambda = \sqrt{\frac{(N-1)3k_B T}{\sum_i mv_i^2}}, \tag{1.16}$$

where λ is the rescaling factor to be applied to the velocities. Note that the previous 2 equations are not in dimensionless units.

1.6 Mean-squared displacement

For our simulation, we chose to have one observable which is the mean-squared displacement (MSD). One can calculate the mean-squared displacement using the following function:

$$\left\langle \Delta^2 \mathbf{x}(t) \right\rangle = \left\langle [\mathbf{x}(t) - \mathbf{x}(0)]^2 \right\rangle, \tag{1.17}$$

where $\mathbf{x}(0)$ is a suitable starting point.

We are not assuming we are having an ideal gas. This means the above equation cannot be simplified using the assumption that the particles move in straight lines.

When the mean-squared displacement is plotted against time, we could see three different behaviors:
- The mean-squared displacement grows quadratically with time, which means the particles are in a gas phase.
- The mean-squared displacement grows linearly with time, which means the particles are in a liquid phase.
- The mean-squared displacement is close to constant or shows oscillatory behaviour, which means the particles are in a solid phase.

It is important to track the mean-squared displacement for a long enough time because for very short times $\left\langle \Delta^2 \mathbf{x}(t) \right\rangle \propto t^2$ for gas, liquid and solid phase.

1.6.1 Diffusion

Using the mean-squared displacement, we can find the diffusion coefficient using the following Einstein relation:

$$D = \lim_{t \to \infty} \frac{1}{6t} \left\langle \Delta^2 \mathbf{x}(t) \right\rangle, \tag{1.18}$$

where D is the diffusion coefficient.

1.7 Errors

For the expectation of a physical observable, we are able to use the following equation:

$$\langle A \rangle = \frac{1}{N} \sum_{n>0}^{N} A_n, \tag{1.19}$$

where A is the physical observable and N is the total simulation time. In our case this observable would be the mean-squared displacement. We will be needing this to determine the errors in our observable data, this will be explained next.

1.7.1 Auto-correlation function

Since our simulation will produce time-evolved data, we cannot calculate the standard deviation through regular means. This is because time-evolved data is not uncorrelated/independent random data. Therefore, the auto-correlation function will be needed in order to determine the standard deviation. The normalized auto-correlation function (also known as the Pearson correlation coefficient) is given as:

$$\chi_A(t) = \frac{1}{\sigma_A^2} \sum_{n} (A_n - \langle A \rangle)(A_{n+t} - \langle A \rangle). \tag{1.20}$$

The auto-correlation function usually shows an exponential decay which is given by the following function:

$$\chi_A(t) = e^{-t/\tau}, \tag{1.21}$$

where τ is the correlation time (referring back to index n). When the value of the correlation time τ is known, we can calculate the error of our simulation observable as follows:

$$\sigma_A = \sqrt{\frac{2\tau}{N}(\langle A^2\rangle - \langle A\rangle^2)}. \tag{1.22}$$

Unfortunately, the auto-correlation function mentioned above is only valid for infinitely long time-evolved data. Since in our simulation, we are restricted to finitely long time-evolved data, the auto-correlation function becomes:

$$\sigma_A = \frac{(N-t)\sum_n A_n A_{n+t} - \sum_n A_n \cdot \sum_n A_{n+t}}{\sqrt{(N-t)\sum_n A_n^2 - (\sum_n A_n)^2}\sqrt{(N-t)\sum_n A_{n+t}^2 - (\sum_n A_{n+t})^2}}. \tag{1.23}$$

1.7.2 Data blocking

The idea of data blocking is to take averages from data blocks. A data block is a chunk of the complete data. When a block length is larger than the correlation time, the block averages become statistically independent random variables. This will solve the before mentioned problem of correlated data, since the averages of the data blocks are now uncorrelated. Therefore, we can now compute the error (standard deviation) through regular means as follows:

$$\sigma_A(b) = \sqrt{\frac{1}{N_b - 1}(\langle a^2\rangle - \langle a\rangle^2)}, \tag{1.24}$$

where b is data block size, N is the total data size and $N_b = N/b$. The averages a_i, which are needed for the previous equation, are calculated as follows:

$$a_i = \frac{1}{b}\sum_{(i-1)*b+1}^{i*b} A_i. \tag{1.25}$$

1.7.3 Block bootstrap

Another method to determine the error for an observable is using the bootstrap method. This is done by creating a new data-set from the original data-set by random picking. However, this does not mean that the new data-set is just a reshuffled version of the original data-set. Instead, every value is randomly picked from the original data-set into the new data-set. This allows the same data-points to be re-picked again as well, and also allows some data-points to not be picked at all. From this new data-set, the observable A can be calculated through regular means again. This process is repeated n times such you will obtain a set of n values for A. The error can now be estimated using:

$$\sigma_A = \sqrt{\langle A^2\rangle - \langle A\rangle^2}, \tag{1.26}$$

where it should be noted that the $1/(n-1)$ term is missing. This is because for a large enough set for A, the error will be independent of the size of the set n. However, it should also be noted that for the bootstrap method statistically independent data is needed for the same reasons as for data blocking: uncorrelated data is a requisite.

1.8 Reproducibility results

Reproducibility of results is self-evident in nearly all disciplines of science. While it is virtually impossible to reproduce the exact initial state of a simulation, the observables of a simulation should be reproducible. Finding (near)optimal, or even usable starting constants is complex, and not the scope of this paper. The box dimensions (L), target temperature (T) and lattice constant L_l are all based the chosen phase of Argon, and related to the the following constants used, based on the book *Computational Physics***source**:

$$\epsilon/k_B = 119.8\text{K}$$

$$\sigma = 3.405 Angstrom$$

$$L_l = 2/L$$

Here, it is important to note that the amount of particles in the simulation is fixed by 32. However, with minimal effort the simulation can also run for $4M^3$ particles, where M is an integer. The parameters for the supposed phases are found in the table down below. It must be noted, that these values are in dimensionless units and designed for 32 particles.

Solid	Liquid	Gas
$T = 0.5$	$T = 1$	$T = 3$
$L = 3.4196$	$L = 3.3130$	$L = 4.7425$

In the simulation, the particles start at a FCC lattice configuration, with the lattice constant being twice the lattice parameter. This is because it is a (approximate) stable configuration, where each particle is spaced evenly, while still adhering the molecular dynamics of Argon. As an initial velocity (also, energy or temperature) a boltzmann distribution has been used, with a fixed temperature of 120K.

1.9 Verification of the results

Phase changes are often related to the following properties: Pressure, temperature, orientation, Verification of the results is vastly complex. For example, pressure is a macroscopic property, and can not be ”measured” from a 32 particle system, but has to be calculated. **source jos thijssen**. Thus, the results consist of the mean squared displacement. However, the simulation can be extended to calculate more observables.

Chapter 2

Results and discussion

2.1 Results

First of all, we will show that the simulation works properly by verifying the simulation. Secondly, we will show how we will show how we pre-process our simulation data which we can later use for our observable. Third, we will be showing the mean-squared displacement of the Argon atoms since the mean-squared displacement is our observable. Fourthly, we will be showing how we derived the diffusion coefficient from the mean-squared displacement. And finally, we will show the errors for the values we have found for our observable. For the results, the following dimensionless constants have been used in addition to the values supplied at **section**.

$$\text{simulation steps} = 10000 \text{ or } 30000$$

$$\text{timestep} = 0.004$$

$$\text{rescaling} = \text{True}$$

2.1.1 Verification of simulation

To verify our simulation, we will first compare the results for different implementations of the classical mechanics. Next, we will present the results for a simple simulation. And finally, we will verify the beforementioned results to verify our simulation.

2.1.1.1 Simple simulation results

Here we can find the simple simulation results for 2 Argon particles.

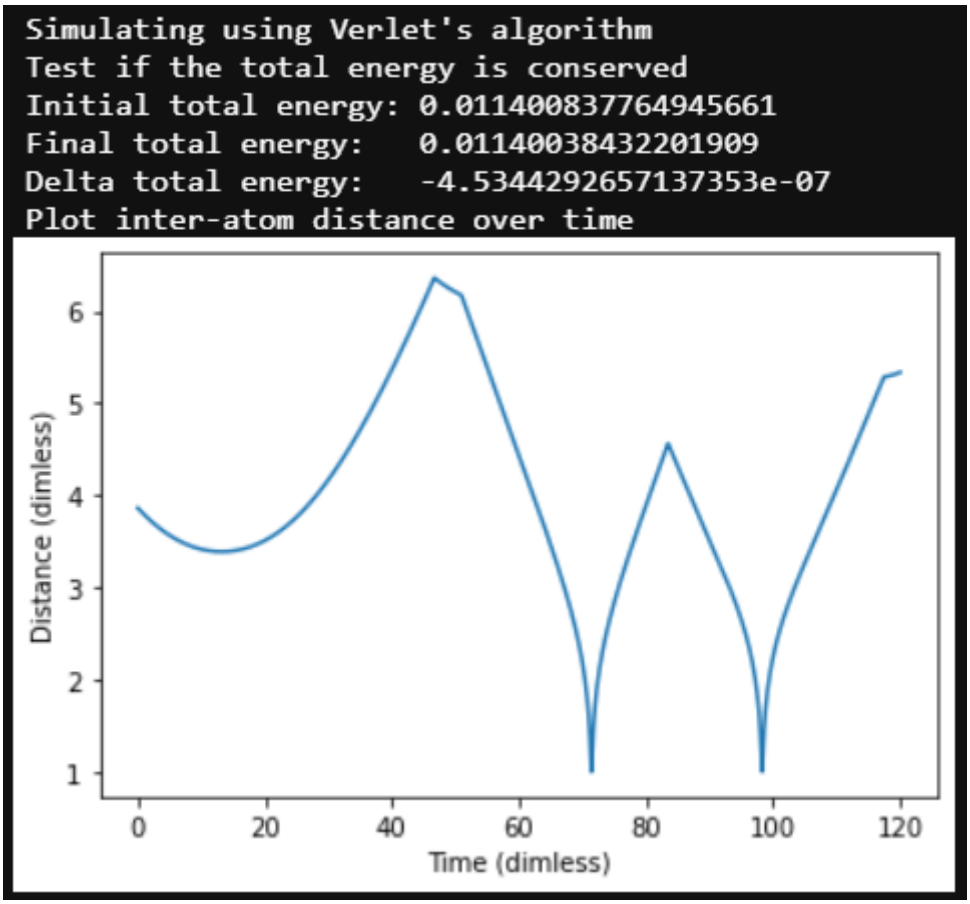

Figure 2.1: Verlet simulation data and inter-atomic distance plot.

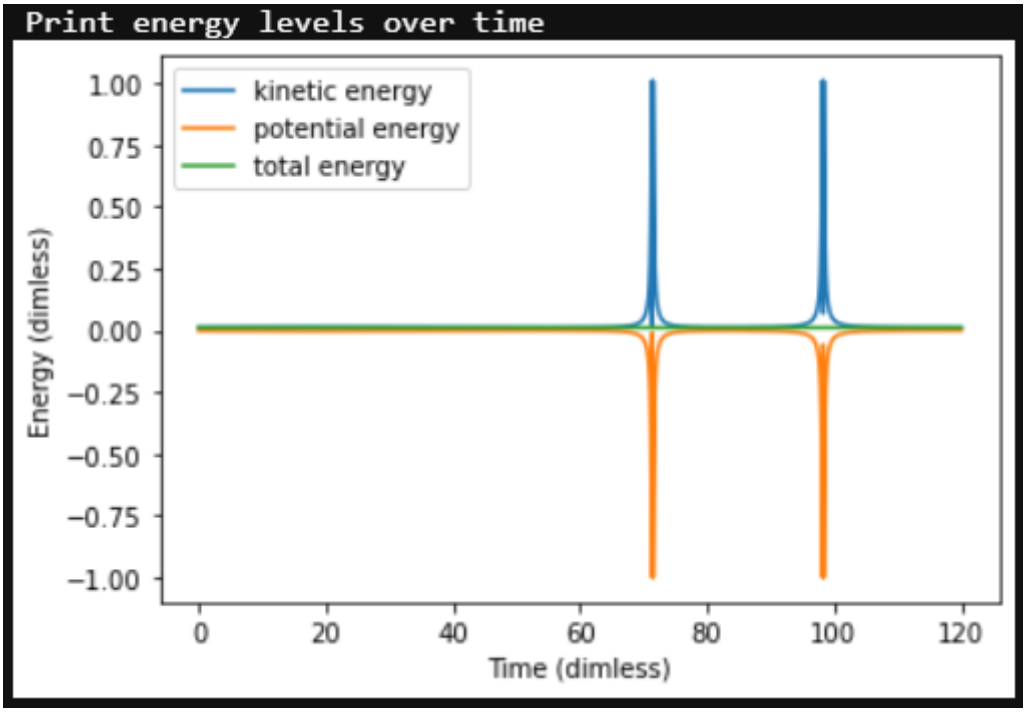

Figure 2.2: Verlet simulation energy plot.

Here, it is shown that in a simple configuration 2 particles will never 'touch', keeping a minimum distance of around 1. This is in line with the Lennard Jones potential due to the powers. Energy is also conserved; from initial to end $4.5 * 10^{-}7$ has been lost, with a with a value of

2.1.1.2 Verlet vs. Euler results

We are comparing the different algorithms to show which algorithm performs better for our simulations and which we will be using to obtain simulation data later.

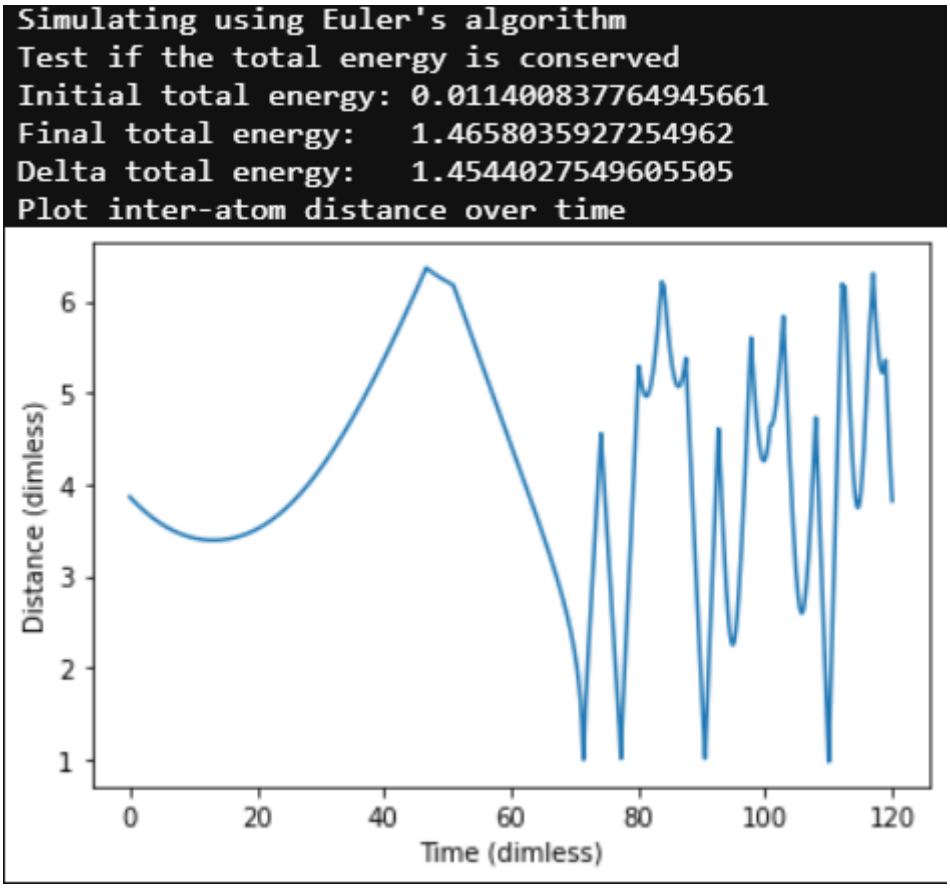

Figure 2.3: Euler simulation data and inter-atomic distance plot.

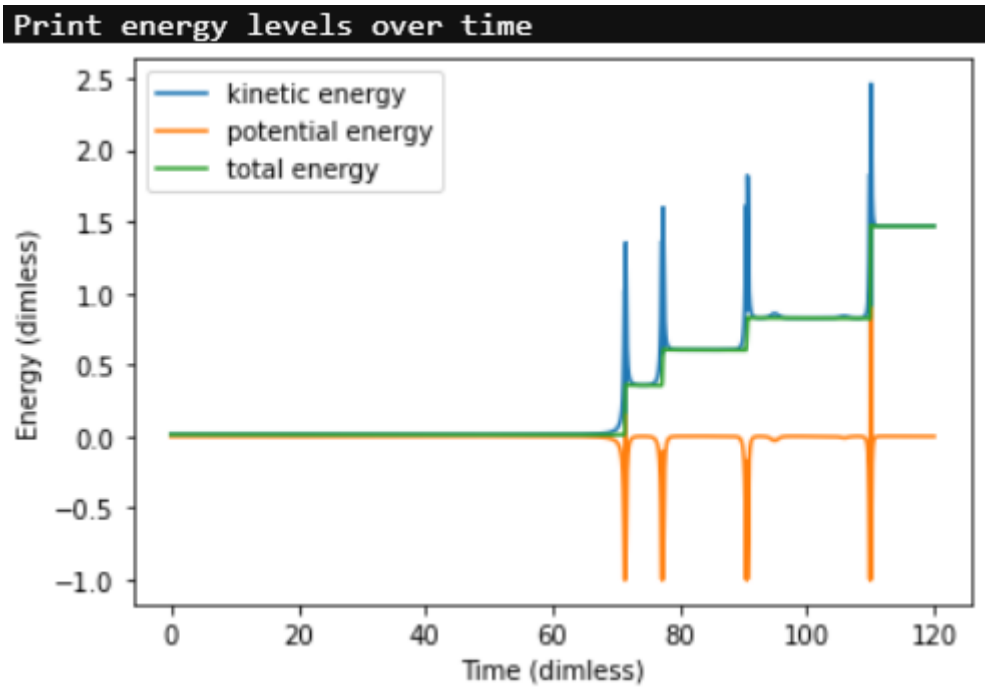

Figure 2.4: Euler simulation energy plot.

It is clear that the Euler method does not conserve energy. For simulations with periodic boundary conditions, this is a problem. The simulation should only change the total energy in the system, if rescaling is applied. The boundary conditions make sure that momentum is conserved; Particles that leave the system are introduced again on the other side of the box, using the nearest-neighbor method, thus every force vector pointing outside the box, has an other vector pointing in the box.**https://aip.scitation.org/doi/10.1063/1.4729** From this moment on, the Euler method shall be disregarded during simulations. It has proven to be an unreliable method to do these types of simulations.

THIS MIGHT BE REMOVED

2.1.1.3 Verification

Here we will verify our simulation. There are multiple things that can be tested to verify a simulation:
- Conservation of energy
- Center of mass stays the same
- Manual approximation of the potential and comparing that to the simulation
- Kinetic energy does not converge after the last rescale

We were unable to test all of these requirements, it can be noted that total energy can be observed. however we do have some other mentionable results.

We were unable to test all of these requirements, but regarding the conversation of energy, it is visible in the plots that this has been conserved. The discrete energy shift of the total energy are due to rescaling, and to be expected. Although the total energy does not seem to converge to an energy, this is to be expected since the rescaling is based on a average, kinetic energy. The final calculated temperature is 0.5069, which is close to the energy set by the rescaling (0.5). The potential energy does not change over time much, however this is due to the chosen box size. **THIS MIGHT BE REMOVED**
We will be showing that the conversation of energy holds for our simulation.

2.1.2 Pre-process simulation data

It is important our velocities are in compliance with the Maxwell-Boltzmann distribution. Since our velocities are chosen to be distributed using a normal/Gaussian distribution, we will receive the following distribution of random velocities:

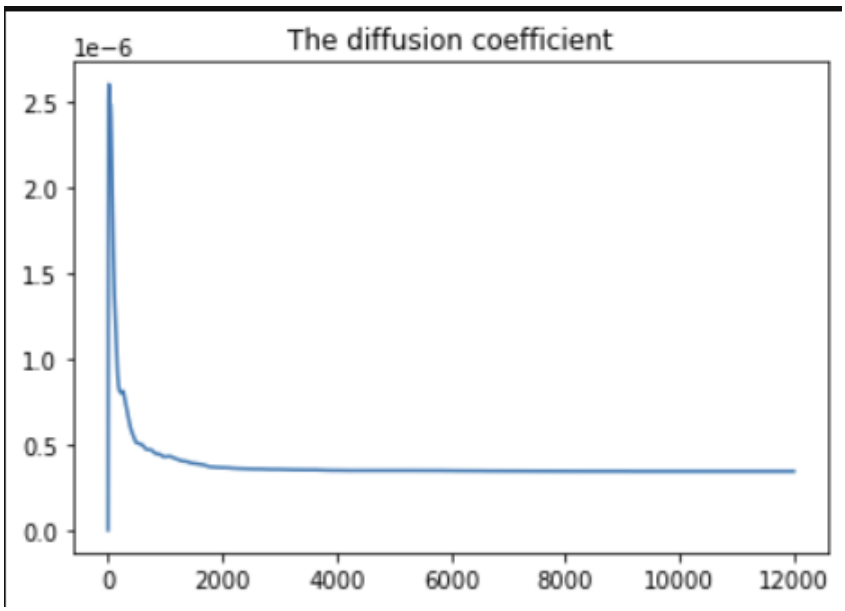

Figure 2.5: Histogram and Gaussian plot for the particle velocities.

As mentioned in the theory, our simulation will use periodic boundary conditions. This will affect the positions from our simulation as they will be stored in a way that it might seem that the movement of the particles over time is not in a continuous motion. To correct this, the periodic particle positions need

to be converted back to non-periodic particle positions. This will be required for the mean-squared displacement, which will be explained next subsection. Below we can find how the non-periodic particle positions over time.

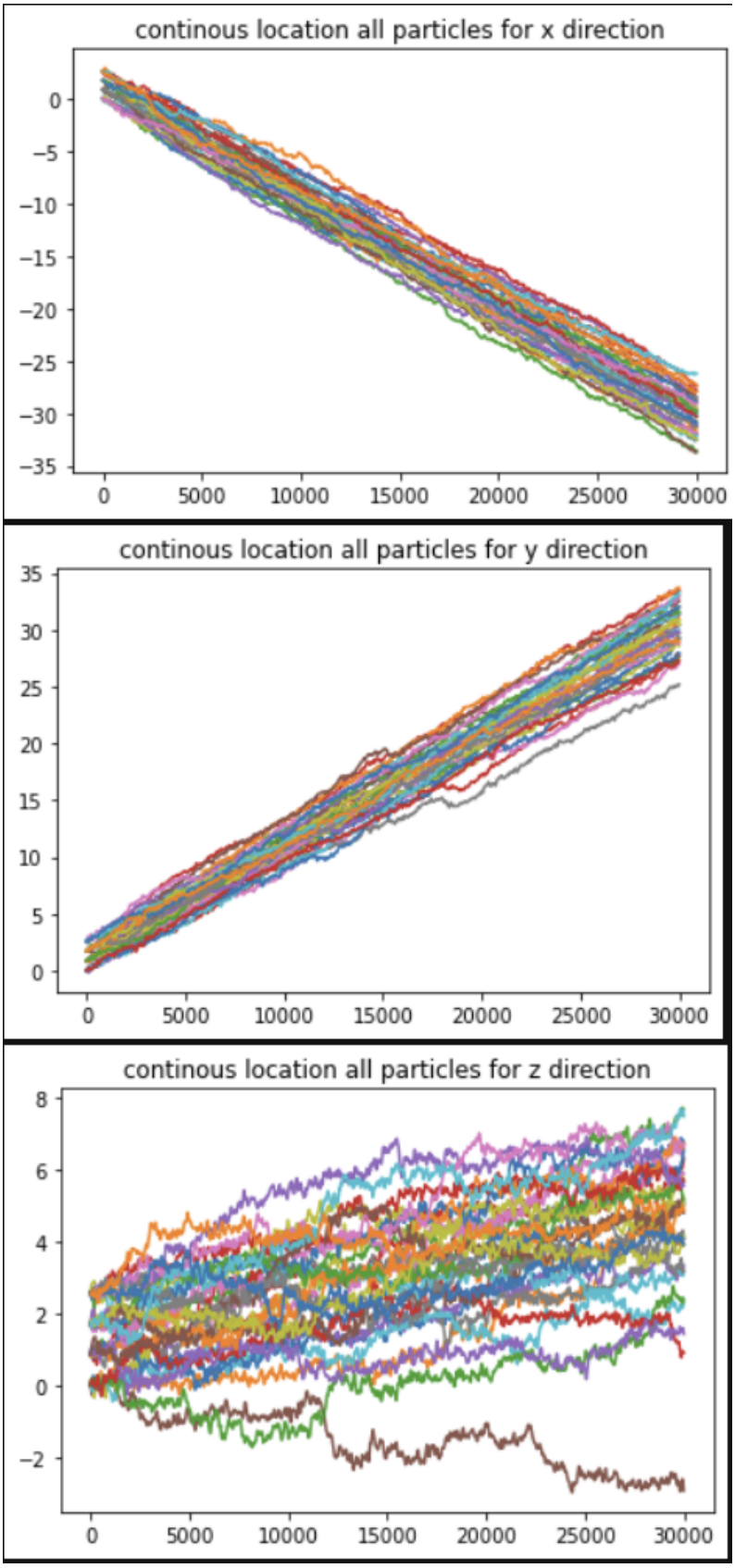

Figure 2.6: Non-periodic particle positions over time for x, y and z coordinates, respectively.

In 2.6 the location is shown. Here something can be noticed, there seems to be a drift. While the Z Below we can find the a plot of the Verlet simulation over 30000 timesteps.

```
Simulating using Verlet's algorithm
Rescaled 18 times with λ: [ 0.9528696321477059 ~ 1.0314650258686826 ]
Last temperature:  0.5069800284321757 ( 60.73620740617465 K )
Test if the total energy is conserved
Initial total energy: -143.62166958751237
Final total energy:   -146.75867670633613
Delta total energy:   -3.1370071188237603
Print energy levels over time
```

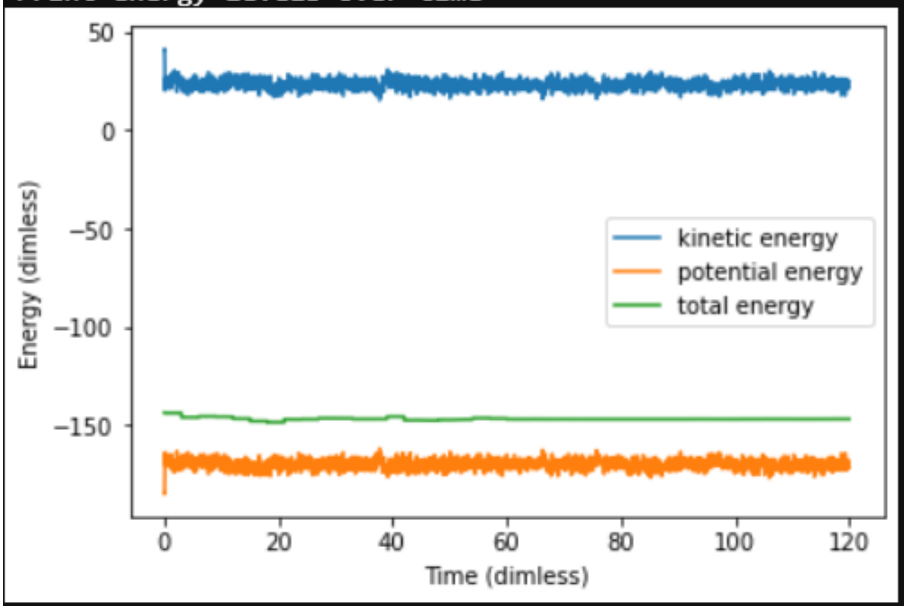

Figure 2.7: Verlet simulation data and energy plot.

In 2.7 the velocities has been rescaled

2.1.3 Mean-squared displacement

Using the non-periodic particle positions from the simulation data, we are able to calculate the mean-squared displacement. In the following plot, the mean squared displacement of the particles is shown.

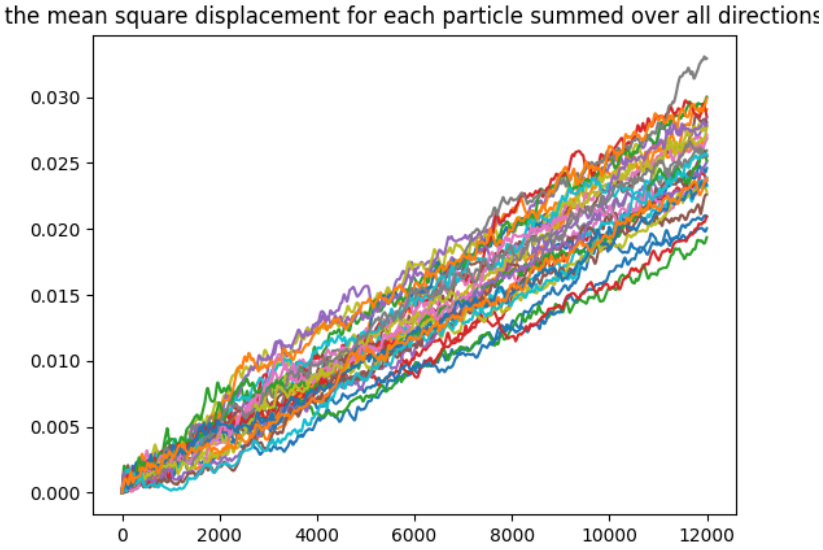

Figure 2.8: Mean-squared displacement of particles over time.

2.1.4 Autocorrelation function

The autocorrelation function is shown in the figure below.

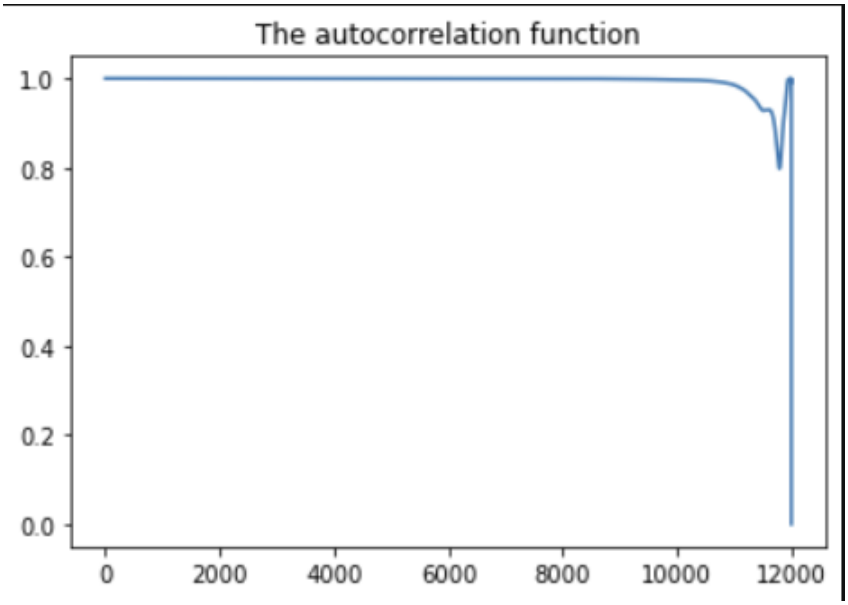

Figure 2.9: Auto-correlation function.

The autocorrelation does not seem to go to 0.

2.1.5 Diffusion coefficient

From the mean-squared displacement, we are able to estimate the diffusion coefficient. This is shown below.

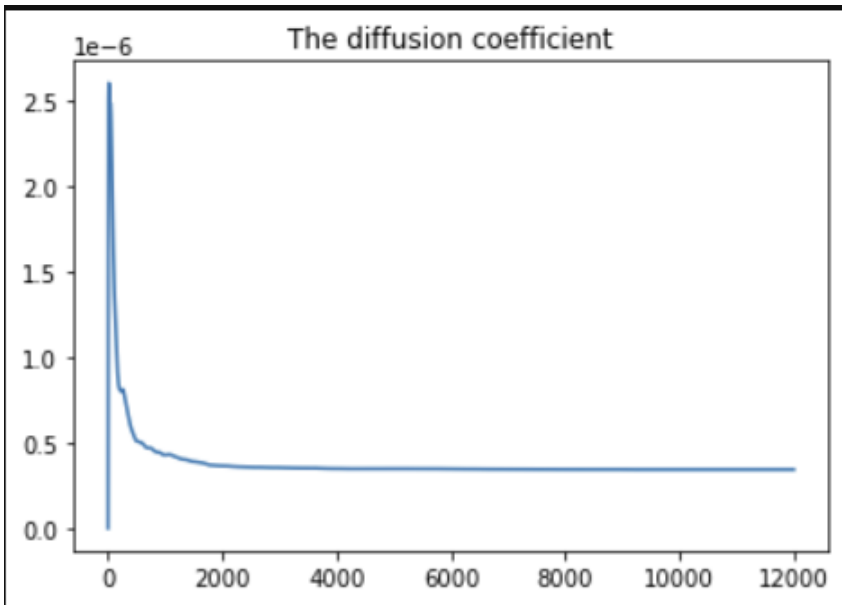

Figure 2.10: Auto-correlation function.

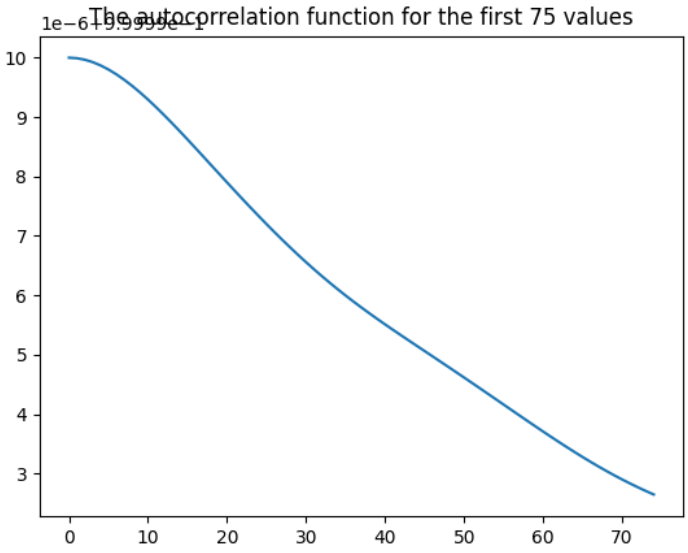

Figure 2.11: Auto-correlation function (cut off).

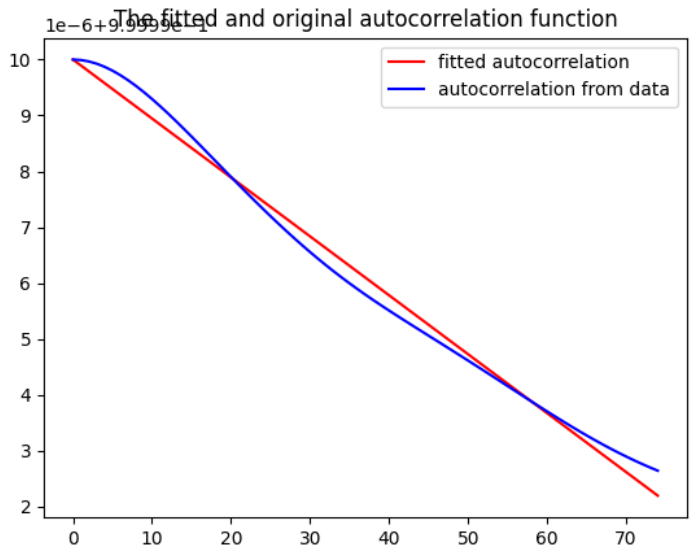

Figure 2.12: Auto-correlation function (cut off) and fitted function.

Tau is 9613606.87407486 and the covariance of Tau is [3.17682736e+09]

2.1.6 Observable errors

Now that the diffusion coefficient has been found from the mean-squared displacement, we should also account for errors in the found values.

The

2.2 Discussion

In this part we will discuss about our results and the overall project. We will mention things that went good, things that went bad and things that could have been improved.

2.2.1 Discussion script/program

2.2.1.1 Efficiency

The efficiency of the program used in this paper is sub-optimal. Some of the most mayor points will be listed below:

2.2.1.1.1 The number of loops present

In the script, the potential energy is calculated after a simulation is completed, using the previously calculated values for the locations. Since the calculation of the potential energy requires the inter-atomic distances, and this is a relative time consuming function, a method should have been implemented to remove the need for this calculation (especially since the amount of computations scales quadratic with the amount of particles). More importantly, the inter-atomic distances were already calculated, however not stored. Due to the **ff een reden waarom dit zo is**

2.2.1.1.2 Compiling

Python has a significant decrease in efficiency when code is not compiled beforehand. Some major improvements can be used by using vectorized functions, using the NumPy module, and then implementing the timestep iteration in a separate function. The usage of vectorized functions itself does probably not decrease efficiency, however it makes the code much easier to debug and/or verify. Which creates the

oppurtunity to rewrite the timestep iteration function for calculating positions and velocities for example such that this function can be compiled beforehand. **niet zeker of dit klopt, maar volgen smij wel. Namelijk onze simulate doet heel veel tegelijk, en niet netjes daarom compiled die anders.**

2.2.1.2 Chosen parameters

2.2.2 Results discussion

In this part we will discuss about the results that were found. We will also compare the values we have found to see if it matches values from literature and values from prior studies.

2.2.2.1 Positive evaluation

When looking at our results we believe that certain things went well. For example...

2.2.2.2 Negative evaluation

However, some things we believe did not go too well. For instance...

2.2.2.3 Points of improvement

Keeping the previous evaluation in mind, here is our list of things we believe could have improved:
- ...
- ...

2.2.3 Project discussion

In this part we will discuss about the overall project went. We will focus on the coding, the repository, the time-management, the (individual) work-load and the communication.

2.2.3.1 Positive evaluation

The overall project went well in our opinion. The communication was good, we were in touch regularly on WhatsApp and on Discord. The division of work-load was fair for the both of us as well. The work was evenly split among us, which would be about 50/50. The GitLab repository was kept tidy at all times and we were concise with our weekly progress report. We used the original 'skeleton.py' file for the coding. We have tried to maintain a similar coding style. This also includes comments and documentation. We also believe we have written quite efficiently running code since our simulations run fast, although we see many points of improvement. We have also actively tried to improve the performance of our code by, for instance replacing lists by NumPy arrays.
A lot of new insights have been obtained, especially for the final steps of a project since there is a big difference between running seperate functions in sequence creating a figure and creating 1 function that does it all in a optimal way.

2.2.3.2 Negative evaluation

Things that could have gone better

2.2.3.3 Points of improvement

Chapter 3

Conclusion

Bibliography

Appendix

A. Python code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
import time
# LOAD SEED
from pickle import load

import simulate as sim
import utilities as utils

with open('state.obj', 'rb') as f:
    np.random.set_state(load(f))


def main():
    """
        Beginning of program
    """

    # Start timer for program run-time
    start_time = time.time()

    # Compare Verlet vs. Euler using simple simulation
    # Simulation parameters
    sim.dim = 3
    sim.box_dim = 9
    sim.num_atoms = 2
    sim.rescaling = False

    # Simple positions and velocities
    init_pos = np.array([[+0.300, +0.600, +0.900], [+2.400, +6.900, +8.100]])
    init_vel = np.array([[-0.090, -0.060, -0.030], [+0.030, +0.060, +0.090]])

    # Verlet
    sim.verlet = True
    pv, vv = sim.simulate(init_pos, init_vel, sim.steps, sim.dt, sim.box_dim)
    utils.process_data(pv, vv)

    # Euler
    sim.verlet = False
    pe, ve = sim.simulate(init_pos, init_vel, sim.steps, sim.dt, sim.box_dim)
    utils.process_data(pe, ve)

    # Keep Verlet for further simulations
```

```
    sim.verlet = True
    # Simulation parameters
    sim.dim = 3
    sim.box_dim = 3.41995
    sim.num_atoms = 32
    sim.rescaling = True
    sim.temp = 0.5 * sim.EPSILON / sim.KB

    # For 32 particles
    # Lattice constant = box_dim/2
    # liquid
    # box_dim = 3.313
    # T = 1

    # solid
    # box_dim = 3.41995
    # T = 0.5

    # Gas
    # 4.7425
    # T = 3

    # FCC lattice positions and random velocities
    init_pos = sim.fcc_lattice(sim.num_atoms, sim.box_dim / 2)
    init_vel = sim.init_velocity(sim.num_atoms, sim.TEMP, sim.dim)

    # Test if (1000) random velocities are in compliance with Maxwell-Boltzmann distribu
    utils.test_initial_velocities(None)

    # Complex Verlet simulation
    p1, v1 = sim.simulate(init_pos, init_vel, sim.steps, sim.dt, sim.box_dim)
    utils.process_data(p1, v1)

    # Find MSD / Diffusion
    utils.auto_corr2(p1)

    # Errors

    # End timer for program run-time
    print("--- %s seconds ---" % (time.time() - start_time))
    return p1, v1


main()
```

```
# coding: utf-8

"""
This is a suggestion for structuring your simulation code properly.
However, it is not set in stone. You may modify it if you feel like
you have a good reason to do so.
"""

import numpy as np

# initalizing self defined system parameters
num_atoms = 4  # amount of particles
dim = 3  # dimensions
```

```
box_dim = 3.313  # meters; bounding box dimension
dt = 4e-3 # s; stepsize

steps = 10000  # amount of steps
dimless = True  # use dimensionless units
periodic = True  # use periodicity
verlet = True  # use Verlet's algorithm (false: Euler's algorithm)
rescaling = True  # use Temperature rescaling
fcc_lattice = True # use FCC lattice
rescaling_mode = 1  # 0 = kin-NRG-based | temp-based
rescaling_delta = 0.09  # delta for activation of rescaling
rescaling_timesteps = steps / 40  # timesteps interval for rescaling check
rescaling_max_timesteps = steps / 2  # max timesteps for rescaling
rescaling_limit = True  # rescale limit [lower~upper]
rescaling_limit_lower = 0.8
rescaling_limit_upper = 1.25
rescaling_factor = 0.5  # 0.5 = sqrt


# Parameters physical, supplied by course, or related to Argon
TEMP = 119.8  # K
KB = 1.38064852e-23  # m^2*kg/s^2/K
SIGMA = 3.405e-10  # meter
EPSILON = TEMP * KB  # depth of potential well/dispersion energy
temp = 1.0 * EPSILON/KB # K (with dimless temp within brackets)

N_b = 6.02214076e23  # Avogadros number; 1/mol
R = 8.31446261815324  # J/K/mole; universal gas constant
ARG_UMASS = 39.95  # u; atomic mass of argon
ARG_MMASS = ARG_UMASS / 1000  # kg/mol; mole mass of argon
ARG_MASS = ARG_UMASS * 1.6605e-27  # Kg mass of a single atom in Kg

# conversion values for dimensionless units
DIMLESS_TIME = 1.0 / np.math.sqrt((ARG_MASS * SIGMA ** 2 / EPSILON))  # s; dimensionless
DIMLESS_ENERGY = 1.0 / EPSILON  # J; dimensionless energy
DIMLESS_DISTANCE = 1.0 / SIGMA  # m; dimensionless distance
DIMLESS_VELOCITY = 1.0 / np.math.sqrt(EPSILON / ARG_MASS)  # m/s; dimensionless velocity


def init_velocity(number_atoms, temperature, dimensions):
    """
    Initializes the system with Gaussian distributed velocities. This
    init_velocity is loosely based on 3D system, however it will output
    2D just fine, although more pertubated. This function is based a
    simplified Boltzmann distribution, found at:
    https://en.wikipedia.org/wiki/Maxwell%E2%80%93Boltzmann
    _distribution#Typical_speeds
    Parameters
    ----------
    number_atoms : int
        The number of particles in the system.
    temperature : float
        The (unitless) temperature of the system.
    dimensions : int
        The dimensions of the system.

    Returns
    -------
    vel_vec : np.ndarray
```

```
73          Array of particle velocities
74      """
75      vel_p = np.sqrt(2 * KB * temperature / ARG_MASS)
76
77      if dimless:
78          vel_p *= DIMLESS_VELOCITY
79
80      vel_mean = 2 * vel_p / np.sqrt(np.pi)
81      vel_msq = (3 * vel_p ** 2) / 2
82      vel_std = vel_msq - (vel_mean ** 2)
83      vel_vec = np.random.normal(vel_mean, vel_std, (number_atoms, dimensions))
84      vel_mag = np.linalg.norm(vel_vec, axis=1)
85      vel_vec *= vel_mean / np.mean(vel_mag)  # Rescale the magnitudes to match the vel_me
86
87      # create random negativity
88      for v in range(number_atoms):
89          for i in range(dimensions):
90              # either *1 or *-1
91              vel_vec[v, i] *= (1 - 2 * np.random.randint(2))
92
93      vel_vec -= np.mean(vel_vec)  # remove mean for no drift velocity
94
95      return vel_vec
96
97
98  def init_position(number_atoms, box_dimensions, dimensions):
99      """
100     Initializes the system with random positions.
101     This does not require non dimensionalization scaling, since it is
102     not based on physical parameters.
103
104     Parameters
105     ----------
106     number_atoms : int
107         The number of particles in the system.
108     box_dimensions : float
109         The dimension of the simulation box
110     dimensions : int
111         The dimensions of the system.
112
113     Returns
114     -------
115     pos_vec : np.ndarray
116         Array of particle positions
117     """
118     randoms = np.random.random((number_atoms, dimensions))
119     pos_vec = randoms * box_dimensions
120
121     return pos_vec
122
123
124 def simulate(init_pos, init_vel, num_tsteps, timestep, box_dimensions):
125     """
126     Molecular dynamics simulation using the Euler or Verlet's algorithms
127     to integrate the equations of motion. Calculates energies and other
128     observables at each timestep.
129
130     Parameters
131     ----------
```

```
    init_pos : np.ndarray
        The initial positions of the atoms in Cartesian space
    init_vel : np.ndarray
        The initial velocities of the atoms in Cartesian space
    num_tsteps : int
        The total number of simulation steps
    timestep : float
        Duration of a single simulation step
    box_dimensions : float
        Dimensions of the simulation box

    Returns
    -------
    Any quantities or observables that you wish to study.
    """

    if verlet:
        print("Simulating using Verlet's algorithm")
    else:
        print("Simulating using Euler's algorithm")

    # total for positions and velocities
    pos_steps = np.zeros((num_tsteps, num_atoms, dim))
    vel_steps = np.zeros((num_tsteps, num_atoms, dim))

    # initial position and velocity
    pos_steps[0, :, :] = init_pos
    vel_steps[0, :, :] = init_vel

    # statistics for rescaling
    rescale_counter = 0
    rescale_max = 1.0
    rescale_min = 1.0

    for i in range(num_tsteps - 1):
        pos = pos_steps[i, :, :]

        if verlet:
            rel_pos, rel_dis = atomic_distances(pos, box_dimensions)
            force = lj_force(rel_pos, rel_dis)[1]

            # Keep particle inside box using modulus when periodic
            if periodic:
                if dimless:
                    pos_steps[i + 1, :, :] = (pos + vel_steps[i, :, :] * timestep
                                              + (timestep ** 2) * force / 2) % box_dimen
                else:
                    pos_steps[i + 1, :, :] = (pos + vel_steps[i, :, :] * timestep + (
                            timestep ** 2) * force / 2) % box_dimensions

            else:
                if dimless:
                    pos_steps[i + 1, :, :] = (pos + vel_steps[i, :, :] * timestep + (tim
                else:
                    pos_steps[i + 1, :, :] = (pos + vel_steps[i, :, :] * timestep + (tim

            # force after position update (needed for verlet velocity)
            new_rel_pos, new_rel_dis = atomic_distances(pos_steps[i + 1, :, :], box_dime
            new_force = lj_force(new_rel_pos, new_rel_dis)[1]
```

```
191
192             if dimless:
193                 vel_steps[i + 1, :, :] = vel_steps[i, :, :] + timestep * (new_force + fo
194             else:
195                 vel_steps[i + 1, :, :] = vel_steps[i, :, :] + timestep * (new_force + fo
196         else:
197             # Euler
198             if periodic:
199                 # make sure it's inside box dimension -> modulus gives periodicity
200                 if dimless:
201                     pos_steps[i + 1, :, :] = (pos + vel_steps[i, :, :] * timestep) % box
202                 else:
203                     pos_steps[i + 1, :, :] = (pos + vel_steps[i, :, :] * timestep) % box
204             else:
205                 if dimless:
206                     pos_steps[i + 1, :, :] = (pos + vel_steps[i, :, :] * timestep)
207                 else:
208                     pos_steps[i + 1, :, :] = (pos + vel_steps[i, :, :] * timestep)
209
210             rel_pos = atomic_distances(pos, box_dimensions)[0]
211             rel_dis = atomic_distances(pos, box_dimensions)[1]
212             force = lj_force(rel_pos, rel_dis)[1]
213
214             if dimless:
215                 vel_steps[i + 1, :, :] = vel_steps[i, :, :] + force * timestep
216             else:
217                 vel_steps[i + 1, :, :] = vel_steps[i, :, :] + force * timestep / ARG_MAS
218
219         if rescaling and (int((i+1) % rescaling_timesteps) == 0) and (i < (rescaling_max
220             # Rescale velocity
221             if rescaling_mode == 0:
222                 # old kin energy avg
223                 rescaling1 = np.sum([kinetic_energy(vel_steps[i - x, :, :])[1] for x in
224                     i + 1, 5000)
225                 # new kin energy avg
226                 rescaling2 = np.sum(
227                     [kinetic_energy(vel_steps[i + 1 - x, :, :])[1] for x in range(min(i
228                 # rescaling factor (in sqrt(...) so values get closer to 1)
229                 v_lambda = np.sqrt((num_atoms - 1) * 3 * KB * temp / (EPSILON * np.sum(
230                     [np.sqrt(np.sum([v[i] ** 2 for i in range(dim)])) for v in vel_steps
231                 current_temperature = rescaling2 * EPSILON / ((num_atoms - 1) * 3 / 2 *
232                 need_rescaling = np.abs(rescaling2 - rescaling1) < rescaling_delta * 0.0
233             else:
234                 # target kin energy
235                 rescaling1 = (num_atoms - 1) * 3 / 2 * temp * KB / EPSILON
236                 # new kin energy avg
237                 kin_nrg = np.zeros(int(min(i + 1, int(rescaling_timesteps))))
238                 for x in range(len(kin_nrg)):
239                     kin_nrg[x] = kinetic_energy(vel_steps[i + 1 - x, :, :])[1]
240                 rescaling2 = np.sum(kin_nrg) / int(min(i + 1, int(rescaling_timesteps)))
241                 v_lambda = np.power(rescaling1/rescaling2,rescaling_factor)
242                 current_temperature = rescaling2 * EPSILON / ((num_atoms - 1) * 3 / 2 *
243                 need_rescaling = np.abs(rescaling2 - rescaling1) > rescaling_delta
244
245             if need_rescaling:
246                 # limit rescaling factor between 0.5 and 2.0
247                 if rescaling_limit:
248                     v_lambda = max(rescaling_limit_lower, v_lambda)
249                     v_lambda = min(rescaling_limit_upper, v_lambda)
```

```
250
251                 # apply rescaling factor
252                 vel_steps[i + 1, :, :] *= v_lambda
253                 # rescaling statistics below
254                 rescale_counter += 1
255                 rescale_max = max(rescale_max, v_lambda)
256                 rescale_min = min(rescale_min, v_lambda)
257
258     if rescaling:
259         # print rescaling statistics
260         print("Rescaled", rescale_counter, "times with λ: [", rescale_min, "~", rescale_
261         print("Last temperature: ", current_temperature*KB/EPSILON, "(", current_tempera
262
263     return pos_steps, vel_steps
264
265
266 def atomic_distances(pos, box_dimensions):
267     """
268     Calculates relative positions and distances between particles.
269
270     parameters
271     ----------
272     pos : np.ndarray
273         The positions of the particles in cartesian space
274     box_dimensions : float
275         The dimension of the simulation box
276
277     returns
278     -------
279     rel_pos : np.ndarray
280         Relative positions of particles
281     rel_dist : np.ndarray
282         The distance between particles
283     """
284
285     dimensions = len(pos[0])
286     # NOTE: includes rel_dist/rel_pos to itself (= 0 / [0.0, 0.0])
287     rel_pos = np.zeros([len(pos), len(pos), dimensions])
288
289     for i in range(0, len(pos)):
290         for j in range(0, len(pos)):
291             for k in range(0, dimensions):
292                 dis = pos[j][k] - pos[i][k]
293                 if periodic:
294                     if dimless:
295                         if dis > (box_dimensions * 0.5):
296                             dis = dis - box_dimensions
297                         if dis <= -(box_dimensions * 0.5):
298                             dis = dis + box_dimensions
299                     else:
300                         if dis > (box_dimensions * 0.5):
301                             dis = dis - box_dimensions
302                         if dis <= -(box_dimensions * 0.5):
303                             dis = dis + box_dimensions
304                 rel_pos[i][j][k] = dis
305
306     rel_dist = np.zeros([len(pos), len(pos)])
307     for i in range(0, len(rel_pos)):
308         for j in range(0, len(rel_pos)):
```

```
309             # before:
310             # rel_dist[i][j] = np.math.sqrt(sum(i ** 2 for i in rel_pos[i][j]))
311             # print(rel_dist[i][j])
312 
313             # after:
314             for val in rel_pos[i][j]:
315                 rel_dist[i][j] += val**2
316             rel_dist[i][j] = np.math.sqrt(rel_dist[i][j])
317 
318 
319     return rel_pos, rel_dist
320 
321 
322 def lj_force(rel_pos, rel_dist):
323     """
324     Calculates the net forces on each atom.
325 
326     Parameters
327     ----------
328     rel_pos : np.ndarray
329         Relative particle positions as obtained from atomic_distances
330     rel_dist : np.ndarray
331         Relative particle distances as obtained from atomic_distances
332 
333     Returns
334     -------
335     force : np.ndarray
336         The force of atom j, on atom i. Where j are total-1 atoms.
337     force_atom : np.ndarray
338         The net force acting on particle i due to all other particles
339 
340     NOTE: THIS IS HOW INPUT CAN BE FOUND:
341     loc = init_position(num_atoms, box_dim, dim)
342     positions = atomic_distances(loc, box_dim)
343     rel_dist = positions[1]
344     rel_pos = positions[0]
345     """
346     dudt = np.zeros([len(rel_dist), len(rel_dist)])
347     force = np.zeros([len(rel_pos[1]), len(rel_pos[1]), len(rel_pos[0][0])])
348 
349     if dimless:
350         for i in range(0, len(rel_pos[1])):  # particle i
351             for j in range(0, len(rel_pos[1])):  # particle i rel to j (!=i)
352                 if i != j:
353                     dudt[i, j] = -24 * ((2 / (rel_dist[i, j] ** 13)) - (1 / rel_dist[i,
354                         rel_dist[i, j])
355                 else:
356                     dudt[i, j] = 0
357         for i in range(0, len(rel_pos[1])):  # particle i
358             for j in range(0, len(rel_pos[1])):  # particle i rel to j (!=i)
359                 force[i, j, :] = dudt[i, j] * rel_pos[i, j, :]
360 
361     else:
362         for i in range(0, len(rel_pos[1])):  # particle i
363             for j in range(0, len(rel_pos[1])):  # particle i rel to j (!=i)
364                 if i != j:
365                     dudt[i, j] = -24 * EPSILON * (
366                             (2 * SIGMA ** 12 / (rel_dist[i, j] ** 13)) - (SIGMA ** 6 / r
367                                     rel_dist[i, j])
```

```
368                 else:
369                     dudt[i, j] = 0
370         for i in range(0, len(rel_pos[1])):  # particle i
371             for j in range(0, len(rel_pos[1])):  # particle i rel to j (!=i)
372                 force[i, j, :] = dudt[i, j] * rel_pos[i, j, :]
373 
374     # while this looks horrible, and is horrible, it works. However, needs significant c
375 
376     force_atom = np.sum(force, axis=1)
377 
378     return force, force_atom
379 
380 
381 def fcc_lattice(number_atoms, lat_const):
382     """
383     Initializes a system of atoms on an fcc lattice.
384 
385     NOTE CURRENTLY, ONLY WORKS FOR 4 ATOMS
386     Initial vectors are:
387     a1 = [D,0,0]
388     a2 = [0,D,0]
389     a3 = [0,0,D]
390     Here, D is the distance between 2 adjecent corner atoms.
391 
392     lattice basis vectors are:
393     r1 = [0,0,0]
394     r2 = 1/2(a1+a2)
395     r3 = 1/2(a2+a3)
396     r4 = 1/2(a3+a1)
397 
398     FCC lattice is only possible in 3D due to definition of FCC lattice
399 
400     https://solidstate.quantumtinkerer.tudelft.nl/10_xray/ can be used as a reference
401 
402     Parameters
403     ----------
404     number_atoms : int
405         The number of particles in the system
406     lat_const : float
407         The lattice constant for an fcc lattice
408 
409     Returns
410     -------
411     pos_vec : np.ndarray
412         Array of particle coordinates
413     """
414     # placeholder
415     pos_vec = 0
416 
417     if number_atoms >= 4:
418         a = np.array([[lat_const, 0, 0], [0, lat_const, 0], [0, 0, lat_const]])
419         # BZ = int(number_atoms / 4)
420         print('FCC lattice possible; N = multiple of 4')
421         # below is not elegant at all, but it works without writing over complex code fo
422         pos_vec = 0.5 * np.array(
423             [[0., 0., 0.], np.add(a[0, :], a[1, :]), np.add(a[1, :], a[2, :]), np.add(a[
424         # offset can be usefull for plotting purposes. Update required to match boxsize
425         offset = [0.01 * box_dim, 0.01 * box_dim, 0.01 * box_dim]  # NOTE I ADDED OFFSET
426         pos_vec = np.add(pos_vec, offset)
```

```
427         # print(pos_vec)
428         # print(a[0,:])
429         if number_atoms > 4:
430             for i in range(2):
431                 pos_ext = pos_vec + a[i, :]
432                 pos_vec = np.append(pos_vec, pos_ext, axis=0)
433             pos_vec = np.append(pos_vec, pos_vec + a[2, :], axis=0)
434             # print('fcc lattice vector is', pos_vec)
435 
436     else:
437         print('N is not multiple of 4, FCC lattice not possible ')
438         exit()
439     return pos_vec
440 
441 
442 def fcc_lattice_big(number_atoms, lat_const):
443     """
444     Initializes a system of atoms on an fcc lattice.
445 
446     NOTE CURRENTLY, ONLY WORKS FOR 4 ATOMS
447     Initial vectors are:
448     a1 = [D,0,0]
449     a2 = [0,D,0]
450     a3 = [0,0,D]
451     Here, D is the distance between 2 adjecent corner atoms.
452 
453     lattice basis vectors are:
454     r1 = [0,0,0]
455     r2 = 1/2(a1+a2)
456     r3 = 1/2(a2+a3)
457     r4 = 1/2(a3+a1)
458 
459     FCC lattice is only possible in 3D due to definition of FCC lattice
460 
461     https://solidstate.quantumtinkerer.tudelft.nl/10_xray/ can be used as a reference
462 
463     Parameters
464     ----------
465     number_atoms : int
466         The number of particles in the system
467     lat_const : float
468         The lattice constant for an fcc lattice
469 
470     Returns
471     -------
472     pos_vec : np.ndarray
473         Array of particle coordinates
474     """
475     # placeholder
476     pos_vec = 0
477     if number_atoms == 14:
478         a = np.array([[lat_const, 0, 0], [0, lat_const, 0], [0, 0, lat_const]])
479         # BZ = int(number_atoms / 4)
480         print('N = multiple of 4')
481         # below is not elegant at all, but it works without writing over complex code fo
482         pos_vec = 0.5 * np.array(
483             [[0., 0., 0.], np.add(a[0, :], a[1, :]), np.add(a[1, :], a[2, :]), np.add(a[
484         pos_vec_ext = np.array([a[0, :], a[1, :], a[2, :], a[0, :] + a[1, :], a[0, :] +
485                                 a[0, :] + a[1, :] + a[2, :], a[0, :] + 0.5 * (a[1, :] +
```

```
                                    a[1, :] + 0.5 * (a[2, :] + a[0, :]), a[2, :] + 0.5 * (a[
        pos_vec = np.append(pos_vec, pos_vec_ext, axis=0)
    else:
        print('value not 14')
    print('fcc lattice vector is', pos_vec)
    return pos_vec


def kinetic_energy(vel):
    """
    Computes the kinetic energy of an atomic system.

    Parameters
    ----------
    vel: np.ndarray
        Velocity of particle

    Returns
    -------
    ke : float
        The total kinetic energy of the system.
    """

    if dimless:
        velsquared = vel ** 2  # np.power(vel, 2.0)
        vel_summed = np.sum(velsquared, axis=1)
        vel_abs = vel_summed ** 0.5  # np.power(vel_summed, 0.5)
        # the total velocity, of 1 particle stored in an array for each particle.
        # Since a bug was present, This is rewritten in, over simplified steps.
        ke_part = 0.5 * vel_abs ** 2  # np.power(vel_abs, 2)
        ke_total = np.sum(ke_part)
    else:
        ke = 0
        for i in range(0, len(vel)):
            ke += 0.5 * ARG_MASS * np.power(np.math.sqrt(sum(i ** 2 for i in vel[i])), 2
        return ke, ke

    return ke_part, ke_total


def potential_energy(rel_dist):
    """
    Computes the potential energy of an atomic system.

    Parameters
    ----------
    rel_dist : np.ndarray
        Relative particle distances as obtained from atomic_distances
    NOTE!
    pos = init_position(num_atoms, box_dim, dim)
    rel_dist = atomic_distances(pos, box_dim)[1]
    !
    Returns
    -------
    pot_e : float
        The potential energy of a single atom, of each other atom.
    pot_etotal : float
        The potential energy of the atom of all other atoms
        NOTE: RETRIEVE BY print(pot[1])
```

```
    pot_total : float
        The total potential energy of the system
    """
    num_atoms1 = len(rel_dist[0])
    pot_e = np.zeros([num_atoms1, num_atoms1])
    for j in range(0, num_atoms1):
        for i in range(0, num_atoms1):
            if i != j:
                pot_e[i][j] = 4 * EPSILON * ((SIGMA / rel_dist[i][j]) ** 12 - (SIGMA / r
            else:
                pot_e[i][j] = 0
    pot_e_particle = np.sum(pot_e, axis=1)
    pot_total = np.sum(pot_e_particle) / 2

    if dimless:
        for j in range(0, num_atoms1):
            for i in range(0, num_atoms1):
                if i != j:
                    pot_e[i][j] = 4 * ((1 / rel_dist[i][j]) ** 12 - (1 / rel_dist[i][j])
                else:
                    pot_e[i][j] = 0
        pot_e_particle = np.sum(pot_e, axis=1)
        pot_total = np.sum(pot_e_particle) / 2

    return pot_e, pot_e_particle, pot_total


def total_energy(vel, rel_dist):
    """
        Computes the total energy of an atomic system.

        Parameters
        ----------
        vel: np.ndarray
            Velocity of particle
        rel_dist : np.ndarray
            Relative particle distances as obtained from atomic_distances

        Returns
        -------
        float
            The total energy of the system.
        float
            The kinetic energy of the system.
        float
            The potential energy of the system.
        -------
        This is simply potential_energy[2]+kinetic_energy[1]

        """

    kin = kinetic_energy(vel)[1]
    pot = potential_energy(rel_dist)[2]

    return kin+pot, kin, pot
```

```
import simulate as sim
import numpy as np
```

```
import matplotlib.pyplot as plt
from scipy import optimize

def ms_displacement(loc, timestep):
    """
        Computes the mean square displacement of a single atom.

        Parameters
        ----------
        loc: np.ndarray
            locations of particles over time [timestep, particle, dims]
        timestep : int
            the timestep of the particle which is used as initial value

        Returns
        -------
        msd_1: np.ndarray
            The msq time dependent array, for N dimensions and M particles
        msd_2: np.ndarray
            the msq time dependent array, summed over the dimensions, for M particles
            [msd_part1(dtime=0), msd_part2(dtime=0),... ], [msd_part1(dtime=1), msd_part
        msd_3: np.ndarray
            the msq time dependent vector, summed over both dimensions and particles
            [msd_total(dtime=0), msd_total(dtime=1),.....]
        D: Diffusion coefficient according to lecture notes. NEEDS ELABORATION
        -------

        """

    # make positions continuous
    displacement = 0.0
    # make array with same size
    p00 = np.zeros_like(loc)
    # time iteration
    for k in range(len(loc[0, 0, :])):
        for j in range(len(loc[0, :, 0])):
            for i in range(len(loc[:, 0, 0])):
                p00[i, j, k] = loc[i, j, k] + displacement
                # last value check
                if i != len(loc[:, j, k]) - 1:
                    # check for discontinuity
                    if loc[i + 1, j, k] > loc[i, j, k] + sim.box_dim / 2:
                        displacement -= sim.box_dim
                    if loc[i + 1, j, k] + sim.box_dim / 2 < loc[i, j, k]:
                        displacement += sim.box_dim
            displacement = 0.0

    init_loc = p00[timestep, :, :]
    # print('currently we take an average for initial location!
    # see diff coefficient and autocorr function to see if it makes sense')
    # init_loc = np.mean(p00[int(timestep*0.99):int(timestep*1.01), :, :], axis=0)
    print(init_loc)
    loc_usage = p00[timestep:-1, :, :]
    msd_1 = np.abs((loc_usage - init_loc) ** 2)
    # next
    for i in range(len(loc_usage[:, 0, 0])):
        msd_1[i, :, :] = msd_1[i, :, :] / (i + 1)

    msd_2 = np.sum(msd_1, axis=2)
```

```
    number_particles = len(loc[0, :, 0])  # number of particles
    msd_3 = np.sum(msd_2, axis=1) / number_particles
    print(len(msd_3))
    diffusion = np.empty(len(msd_3))
    for i in range(len(msd_3)):
        diffusion[i] = msd_3[i] / (6 * (i + 1))

    plt.plot(p00[:, :, 0])
    plt.title('continous location all particles for x direction')
    plt.show()
    plt.plot(p00[:, :, 1])
    plt.title('continous location all particles for y direction')
    plt.show()
    plt.plot(p00[:, :, 2])
    plt.title('continous location all particles for z direction')
    plt.show()
    plt.plot(msd_2[:, :])
    plt.title('the mean square displacement for each particle summed over all directions
    plt.show()
    plt.plot(diffusion)
    plt.title('The diffusion coefficient')
    plt.show()
    # print('the diff coeff is shown in the plot above', D)
    return msd_1, msd_2, msd_3, diffusion


# +
# q = ms_displacement(program[0], 15000)
# plt.plot(program[0][:,:,2])
# plt.plot(q[0][:,:,2])
# plt.plot(q[3])

# +
# plt.plot(q[3])
# -

def msd_plot(msd, partnum):
    """
    plots the MSD of a single atom NOTE MIGHT NOW WORK
    Parameters
    ----------
    msd: np.ndarray
        the msq time dependent array, summed over the dimensions, for M particles
        [msd_part1(dtime=0), msd_part2(dtime=0),... ], [msd_part1(dtime=1), msd_part2(ti
        best use case: msd_2 from ms_displacement function
    partnum: int
        the particle that is to be plotted by the function
    Returns
    ----------
    None
    """

    plt.plot(msd[:, partnum])
    plt.show()
    return


def auto_corr(data_values, skipvalues):
    """
```

```
121     gives the normalized autocorrelation function of an obersvable function.
122
123     Parameters
124     ---------------
125     data_values: np.ndarray 1D
126         The data values used corresponding to the expectation value. This should be an 1
127         most likely, this is ms_deviation[2]
128     skipvalues: int
129         skips these initial values. NOTE KEEP AT 0 FOR REPORT.
130     Returns
131     -------------
132     Autocorrelation: np.ndarray
133         The autocorrelation function for t
134     """
135     data_values = data_values[skipvalues:-1]
136     number_particles = len(data_values)
137     autoc = np.zeros(number_particles)
138     for i in range(number_particles - 1):
139         nmax = number_particles - i
140         ant = data_values[i:number_particles:1]
141         an = data_values[0:nmax:1]
142         autoc[i] = ((number_particles - i) * np.sum((ant * an)) - (np.sum(an) * np.sum(a
143                 np.sqrt((number_particles - i) * np.sum(an ** 2) - np.sum(an) ** 2) * np
144                     (number_particles - i) * np.sum(ant ** 2) - np.sum(ant) ** 2))
145     plt.plot(autoc)
146     plt.title('The autocorrelation function')
147     plt.show()
148     return autoc
149
150
151 def auto_corr2(data):
152     # plot the autocorrelation function (NEEDS TO BE MOVED TO utililities.py)
153     q = ms_displacement(data, int(sim.rescaling_max_timesteps * 1.2))
154     focusdiff = 0
155     plt.title(('The Diffusion coefficient skipping the first', str(focusdiff), 'values')
156     plt.plot(q[3][focusdiff:])
157     plt.show()
158     qq = auto_corr(q[2], 0)
159     plotfocus = 75 #sim.steps/50
160     plt.plot(qq[0:plotfocus])
161     plt.title(('The autocorrelation function for the first ' + str(plotfocus) + ' values
162     plt.show()
163     exponential_fit(qq, plotfocus)
164     plt.plot(q[1][0:300])
165     plt.show()
166
167
168 def exponential_fit(y_data, cutoff):
169     """"
170     Gives exponential fit of ydata given, removing everything after the cutoff index.
171     Note: does not use initial guesses. Check manually from graph if it is okay.
172
173     Parameters
174     ---------------
175     y_data: np.ndarray 1D
176         The data that is to be fitted.
177     cutoff: int
178         the last datapoint of ydata that is to be used.
179
```

```
    Returns
    -------------
    Params, Tau: float
        fit parameters of the exponential fit
    params_covariance, Covarance of Tau: float
        covariance of tau
    All return values are only taking the data before the cutoff y_data
    """
    numpoints = len(y_data[0:cutoff])
    x_data = np.linspace(0, numpoints, num=numpoints)

    def funcexp(x, tau):
        return np.exp(-x / tau)

    params, params_covariance = optimize.curve_fit(funcexp, x_data, y_data[0:cutoff])
    print('Tau is ', params[0], 'and the covariance of Tau is', params_covariance[0])

    plt.plot(funcexp(x_data, params[0]), 'r', label='fitted autocorrelation')
    plt.plot(y_data[0:cutoff], 'b', label='autocorrelation from data')
    plt.title('The fitted and original autocorrelation function')
    plt.legend()
    plt.show()
    return params, params_covariance


# +
# plt.plot(Q)
# plt.plot(Q[0:4000])
# -
def process_data(positions, velocities):
    print("Test if the total energy is conserved")
    pos1 = positions[0, :, :]
    pos2 = positions[sim.steps - 1, :, :]

    vel1 = velocities[0, :, :]
    vel2 = velocities[sim.steps - 1, :, :]

    r_pos1 = sim.atomic_distances(pos1, sim.box_dim)
    r_pos2 = sim.atomic_distances(pos2, sim.box_dim)

    print("Initial total energy: " + str(sim.total_energy(vel1, r_pos1[1])[0]))
    print("Final total energy:   " + str(sim.total_energy(vel2, r_pos2[1])[0]))
    print("Delta total energy:   " + str(sim.total_energy(vel2, r_pos2[1])[0] - sim.tota

    times = np.linspace(0, sim.dt * sim.steps, sim.steps)
    if sim.num_atoms == 2:
        print("Plot inter-atom distance over time")
        distances = np.zeros(sim.steps)
        if sim.dimless:
            for x in range(sim.steps):
                distances[x] = np.max(sim.atomic_distances(positions[x, :, :], sim.box_d
            # distances = [np.max(atomic_distances(positions[x, :, :], box_dim)[1]) for
        else:
            distances = [np.max(sim.atomic_distances(positions[x, :, :], sim.box_dim)[1]
        plt.plot(times, distances)
        if sim.dimless:
            plt.ylabel('Distance (dimless)')
            plt.xlabel('Time (dimless)')

```

```
239         else:
240             plt.ylabel('Distance (m)')
241             plt.xlabel('Time (s)')
242 
243         plt.show()
244 
245     print("Print energy levels over time")
246     energies = np.zeros([3, sim.steps])
247     if sim.dimless:
248         for x in range(sim.steps):
249             t, k, p = sim.total_energy(velocities[x, :, :],
250                                        sim.atomic_distances(positions[x, :, :], sim.b
251             energies[0, x] = k
252             energies[1, x] = p
253             energies[2, x] = t
254         # energies = [(kinetic_energy(velocities_store[x, :, :])[1],
255         #              potential_energy(atomic_distances(positions_store[x, :, :], box_di
256         #              total_energy(velocities_store[x, :, :],
257         #                           atomic_distances(positions_store[x, :, :], box_dim)[1
258         #            for x in range(steps)]
259         # energies = np.array(energies)
260     else:
261         energies = [sim.kinetic_energy(velocities[x, :, :])[1] for x in range(sim.steps)
262 
263     # times = np.linspace(0, dt*steps, steps)
264     plt.plot(times, energies.T)
265     plt.xlabel('Time (dimless)')
266     plt.ylabel('Energy (dimless)')
267     plt.legend(('kinetic energy', 'potential energy', 'total energy'))
268     plt.show()
269     return energies
270 
271 
272 def locationplot(locations, latmult):
273     """
274     Plots locations of N particles
275 
276 
277     Parameters
278     ----------
279     locations : np.ndarray
280         locations of particles
281     latmult: scalar
282         How many lattices are to be plotted\
283         latmult=1 4particles; latmult=2 16particles
284 
285     Returns
286     -------
287     plot : plt.plot
288         plot of the locations of the particles.
289     """
290     fig = plt.figure()
291     ax = fig.add_subplot(111, projection='3d')
292     ax.scatter(locations[:, 0], locations[:, 1], locations[:, 2])
293     ax.set_xlim3d(0, latmult)
294     ax.set_ylim3d(0, latmult)
295     ax.set_zlim3d(0, latmult)
296     plt.show()
297 
```

```
298
299 # q = fcc_lattice(32,1)
300 # locationplot(q,2)
301
302
303 def test_initial_velocities(init_velocities):
304     if init_velocities is None:
305         init_velocities = sim.init_velocity(1000, sim.TEMP, sim.dim)
306
307     vel_mag = np.linalg.norm(init_velocities, axis=1)
308     # [np.sqrt(np.sum([v[i] ** 2 for i in range(dim)])) for v in init_velocities]
309
310     gaussian_mean = np.mean(vel_mag)
311     gaussian_sigma = np.std(vel_mag)**2
312     gaussian_max = np.max(vel_mag)
313
314     x_axis = np.linspace(0.5,3.0,1000)
315
316     gaussian = np.exp(-np.power(x_axis-gaussian_mean,2.0)/gaussian_sigma/2)
317
318     y, x, _ = plt.hist(vel_mag, bins=15)
319
320     gaussian *= np.max(y)
321     plt.plot(x_axis, gaussian)
322     plt.show()
```

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from scipy import optimize
4
5
6  def normal_autocorr(mu, sigma, tau, N):
7      """Generates an autocorrelated sequence of Gaussian random numbers.
8
9      Each of the random numbers in the sequence of length `N` is distributed
10     according to a Gaussian with mean `mu` and standard deviation `sigma` (just
11     as in `numpy.random.normal`, with `loc=mu` and `scale=sigma`). Subsequent
12     random numbers are correlated such that the autocorrelation function
13     is on average `exp(-n/tau)` where `n` is the distance between random
14     numbers in the sequence.
15
16     This function implements the algorithm described in
17     https://www.cmu.edu/biolphys/deserno/pdf/corr_gaussian_random.pdf
18
19     Parameters
20     ----------
21
22     mu: float
23         mean of each Gaussian random number
24     sigma: float
25         standard deviation of each Gaussian random number
26     tau: float
27         autocorrelation time
28     N: int
29         number of desired random numbers
30
31     Returns:
32     --------
```

```
    sequence: numpy array
        array of autocorrelated random numbers
    """
    f = np.exp(-1./tau)

    sequence = np.zeros(shape=(N,))

    sequence[0] = np.random.normal(0, 1)
    for i in range(1, N):
        sequence[i] = f * sequence[i-1] + np.sqrt(1 - f**2) * np.random.normal(0, 1)

    return mu + sigma * sequence


def auto_corr(data_values, skipvalues):
    """"
    gives the normalized autocorrelation function of an obersvable function.

    Parameters
    ---------------
    data_values: np.ndarray 1D
        The data values used corresponding to the expectation value. This should be an 1
        most likely, this is ms_deviation[2]
    skipvalues: int
        skips these initial values. NOTE KEEP AT 0 FOR REPORT.
    Returns
    -------------
    Autocorrelation: np.ndarray
        The autocorrelation function for t
    """
    data_values = data_values[skipvalues:]
    N = len(data_values)
    autoc = np.zeros(N)
    for i in range(N-1):
        nmax = N - i
        Ant = data_values[i:N:1]
        An = data_values[0:nmax:1]
        autoc [i] = ((N-i) * np.sum( (Ant*An) ) - ( np.sum(An) * np.sum(Ant) )) / ( np.s
    plt.plot(autoc)
    plt.title('The autocorrelation function')
    plt.show()
    return autoc


def exponential_fit(y_data, cutoff):
    """"
    Gives exponential fit of ydata given, removing everything after the cutoff index. No

    Parameters
    ---------------
    ydata: np.ndarray 1D
        The data that is to be fitted.
    cutoff: int
        the last datapoint of ydata that is to be used.

    Returns
    -------------
    Params, Tau: float
        fit parameters of the exponential fit
```

```
92      params_covariance, Covarance of Tau: float
93          covariance of tau
94      All return values are only taking the data before the cutoff y_data
95      """
96      numpoints = len(y_data[0:cutoff])
97      x_data = np.linspace(0,numpoints, num=numpoints)
98      def funcexp(x, tau):
99          return np.exp(-x/tau)
100
101     params, params_covariance = optimize.curve_fit(funcexp, x_data, y_data[0:cutoff])
102     print('Tau is ', params[0], 'and the covariance of Tau is', params_covariance[0])
103
104     plt.plot(funcexp(x_data, params[0]), 'r', label='fitted autocorrelation')
105     plt.plot(y_data[0:cutoff], 'b', label='autocorrelation from data')
106     plt.title('The fitted and original autocorrelation function')
107     plt.legend()
108     plt.hlines(0, 0, cutoff)
109     plt.show()
110     return params, params_covariance
111
112
113 def expectedvalues(y_data, cutoff):
114     """"
115     Gives expected value according to <A> = 1/N * sum(An) with n>0 as lower boundary, ar
116     Please note, here the index starts at 0, not at 1 as in the literature.
117
118     Parameters
119     ---------------
120     y_data: np.ndarray 1D
121         The input array, An.
122     cutoff: int
123         the last datapoint of y_data that is to be used.
124
125     Returns
126     -------------
127     expected: float
128         The expected value of y_data <A>
129     squared_expected: float
130         The squared expected value <A**2>
131     expected_squared
132         the squared value of the expected value <A>**2
133     """
134     A = y_data[:cutoff]
135     N = len(A)
136
137     # Python 3.9 fix... (works fine in Python 3.8)
138     #for x in range(len(A)):
139     #    val = A[x]
140     #    if (np.abs(val) > 1) or val == float("inf") or val != val:
141     #        print("Error:", val)
142     #        A[x] = 0
143
144     sumA = np.sum(A)
145
146     expected = 1/N * sumA
147     expected2 = np.power(expected,2.0)
148     square_expected = 1/N * sum(A**2)
149
150     return expected, expected2, square_expected
```

```
151
152
153 def errortau(y_data, tau):
154     """
155     calculates the error in the mean of the autocorrelation function.
156
157     Parameters
158     ---------------
159     ydata: np.ndarray 1D
160         The data that is to be fitted.
161     cutoff: int
162         the last datapoint of ydata that is to be used.
163
164     Returns
165     -------------
166     Params, Tau: float
167         fit parameters of the exponential fit
168     params_covariance, Covarance of Tau: float
169         covariance of tau
170     All return values are only taking the data before the cutoff y_data
171     """
172     N = len(y_data)
173     expectedA = expectedvalues(y_data, N)
174     sigma = expectedA[2] - expectedA[1]
175     sigmaA = np.sqrt(2*tau*sigma/N)
176     return sigmaA, sigma
177
178
179 def block_data(y_data, block_length):
180     """
181     Takes average of the block_length values, and puts this in a array.
182
183     Parameters
184     ---------------
185     y_data: np.ndarray 1D
186         The input array that requires data blocking.
187     block_length: integer
188         The required block length.
189
190     Returns
191     -------------
192     a: np.ndarray 1D
193         The new array of the block, created by taking the block averaged of the y_data.
194     """
195     Nb = len(y_data)//block_length
196     a = np.empty(Nb)
197     for i in range(1,Nb):
198         a[i] = sum(y_data[((i-1)*block_length)+1:i*block_length])/block_length
199     np.delete(a, 0)
200     return a
201
202
203 def errorblock(meanblocks):
204     """
205     Calculated the error of the mean, taking the datablocks as input. These datablocks c
206     Note: This is dependent on the size of the block!
207
208     Parameters
209     ---------------
```

```
    meanblocks: np.ndarray 1D
        Input array, in the literature of this course called a_i
    y_data: np.ndarray 1D
        The input array that requires data blocking.
    block_length: integer
        The required block length.

    Returns
    -------------
    sigmaAb: np.ndarray 1D
        the standard deviation of the estimator of the mean (error of the mean).
    """
    expecteda = expectedvalues(meanblocks, len(meanblocks))
    sigmaAb = np.sqrt((expecteda[2]-expecteda[1])/(len(meanblocks)-1))
    return sigmaAb


def error_mean(y_data, cutoff):
    """
    Calculates the error in the mean of the observable
    Shows the plot of the autocorrelation function.
    To verify results:
    Check where the errorvsblocksize converges.

    Parameters
    ---------------
    y_data: np.array 1D
        observable
    cutoff: int
        the cutoff value determined by the autocorrelation function.

    Returns
    -------------
    none
    """


    autofun = auto_corr(y_data, 0)
    fit = exponential_fit(autofun, cutoff)
    max_block_size = int(len(y_data)/25)
    errora = np.empty(max_block_size)
    for i in range(2,max_block_size):
        blocks = block_data(y_data, i)
        errora[i] = errorblock(blocks)
    plt.plot(errora)
    plt.title('error vs block size')
    plt.show()
    tauer = errortau(y_data, fit[0])
    print('uncertainty in the mean is ', tauer[0])
    return


# +
tau = 50
mu = 0
sigma = 1
N = 20000
```

```
y_data = normal_autocorr(mu, sigma, tau, N)
autofun = auto_corr(y_data, 0)
plt.plot(y_data)
plt.show()


# -

fit = exponential_fit(autofun, 300)

max_block_size=300
errora = np.empty(max_block_size)
for i in range(2,max_block_size):
    blocks = block_data(y_data, i)
    errora[i] = errorblock(blocks)
plt.plot(errora)
plt.show()
tauer = errortau(y_data, fit[0])
#print(tauer)

y_ata = normal_autocorr(mu, sigma, tau, N)
error_mean(y_ata, 300)
```