Assignment 1

Vladimir Jelle Lagerweij*

May 25, 2022

Abstract

This report is written for assignment 1 of the introduction to molecular simulation course at the mechanical engineering masters track at the TU Delft. The main topic of this report is a python program which simulate a Lennard Jones fluid using a canonical Monte Carlo algorithm. A simulation of liquid and gaseous methane is used to show the performance of this code. A simple TraPPE forcefield is used to model the interactions between the methane molecules. The intramolecular interactions are not taken into account in this force field, which only considers LJ interactions between the molecules. As the goal of this report is to show my personal development and the choices made while coding my program in python, I will keep the language informal.
The resulting code is optimised for ensembles ranging from 250 to 1000 particles with a large cut-off distance. This results in smooth radial distribution functions with long tails. Array operations in numpy are used to increase the computational efficiency as much as possible and the observables -potential energy, pressure and the radial distribution function- are sampled every so often to reduce their effect on the computation time. The example of modelling methane shows that the code is able to accurately predict the pressure in both the liquid and gaseous phase and is able to clearly show the radial distribution function that is expected of these phases. The produced code exists of two files: the class file ('LJ_Monte_Carlo.py'), which is the general canonical Monte Carlo program, and an example execute file ('execute.py') that shows the practical use of the class file for methane.

Keywords: Homework, Self-study, Monte Carlo simulation, methane, computational modelling

1 Setting up the code

In the following section, some physics needed for the Monte Carlo code is explained, followed by the code used to implement these phenomena. Before these functions are shown, it should be mentioned that I used object oriented coding for the assignment: I use a class to keep track of the state of my ensemble. Therefore, some functions will be implemented slightly different than instructed in the assignment. To show the functions more compact in the report, I decided to remove the comments in this report. The full code, including the comments can be found on my GitHub page [1].

1.a Total energy

To calculate the total energy of the ensemble in a certain configuration can be done by computing the following summation:

$$U_{\text{tot}} = \begin{cases} 4\varepsilon \sum_{i=0}^{N} \sum_{j>i}^{N} \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] & r_{ij} \leq r_{\text{cut}} \\ 0 & r_{ij} > r_{\text{cut}} \end{cases} \tag{1}$$

Where the tail and shift correction can be implemented using the following equations:

$$U_{\text{shift}} = 4\varepsilon \left[\left(\frac{\sigma}{r_{\text{cut}}}\right)^{12} - \left(\frac{\sigma}{r_{\text{cut}}}\right)^{6} \right] \tag{2}$$

$$U_{\text{tail}} = \frac{8}{3}\pi\rho_{\text{N}}\varepsilon\sigma^3 \left[\frac{1}{3}\left(\frac{\sigma}{r_c}\right)^{9} - \left(\frac{\sigma}{r_c}\right)^{3} \right] \tag{3}$$

For correct computations of this, all r_{ij} should be calculated first. As these distances are used for the virial pressure as well, computing them is done in a separated function.

```
1  def allDistances(self):
2          x = self.x
3          L = self.L
4          N, D = x.shape
5
6          r = np.broadcast_to(x, (N, N, D))
7          rel_r = (r - r.transpose(1, 0, 2) + L/2) % L - L/2
8          d_sq = np.einsum('ijk, ijk->ij', rel_r, rel_r,
           ↪  optimize='optimal')
9          np.fill_diagonal(d_sq, np.inf)
10         self.d_sq = d_sq
```

Listing 1: The function which calculates all distances between all particles.

This is a function included in the state, therefore, the only input needed is `self`. Of the state recorded in the class, only the locations of all current particles and the size of the box are needed to compute all distances. I decided to optimise the distance computations. In line 7, the size of the position array is increased to a 3D array, by copying the positional information N times. Subtracting the transpose of this from itself computes the inter particle distances immediately. Using the modulus statement applies the periodic boundary conditions. From these, the square of the norm of these position vectors are calculated. The `numpy.broadcast()` and `numpy.einsum()` are chosen for computational efficiency. The einsum is tested to be faster than `numpy.linalg.norm()` and `numpy.broadcast()` creates the larger array, but changes the memory access instead of creating the full array in the memory.

Now the total energy can be computed. This function needs the imputs σ, ε, r_{cut} and more importantly all the distances computed. Besides that, the shift ant tail corrections (if activated in the code) are included as well.It must be noted that the dis-

* Corresponding author. e-mail: v.j.lagerweij@student.tudelft.nl
[1] Link to GitHub page of this project

tance function only computes the squared distances, as there is no use in computing the root anyway. The code calculates the Van der Waals and Pauli parts separately and then calculates the total energy from combining them. To include the correct tail and shift corrections, the amount of particles within the cut off distance and outside the cut off distance is evaluated.

```
def totalEnergy(self):
    d_sq = np.copy(self.d_sq)
        N = d_sq.shape[0]
        Rcut = self.Rcut
        eps = self.eps
        sig = self.sig
        tail = self.tail
        shift = self.shift

        d_sq[d_sq > Rcut**2] = np.inf
        sr6 = sig**6/(d_sq*d_sq*d_sq)
        sr12 = sr6*sr6
        n = np.count_nonzero(sr6)
        Etot = 2*eps*np.sum(sr12-sr6) - n*shift/2 + N*tail
        self.Etot = Etot
        return Etot
```

Listing 2: The total energy function.

1.b Single particle energy

For the single particle energy, a similar approach is taken. However, as the only use for the distances between one single particle and all the others is that of the single particle energy, the decision is made to combine them into one single function. The inputs for this functions are the current state of the system: The current positions of all particles, the box size, and the constants: ε, σ, and the cut-off distance r_{cut}. The only manual input needed is the index of the particle of which the interaction energy is investigated.

```
def singleEnergy(self, Ni, trial=False):
        sig = self.sig
        eps = self.eps
        tail = self.tail
        shift = self.shift

        if trial is True:
                x = self.x_trial
        else:
                x = self.x

        L = self.L
        Rcut = self.Rcut
        N, D = x.shape

        rel_r = (x - x[Ni, :] + L/2) % L - L/2
        d_sq = np.einsum('ij, ij -> i', rel_r, rel_r)
        d_sq[Ni] = np.inf
        d_sq[d_sq > Rcut**2] = np.inf

        sr_2 = sig**2/d_sq
        sr6 = sr_2*sr_2*sr_2
        sr12 = sr6*sr6

        n = np.count_nonzero(sr6)
        E_single = 4*eps*np.sum(sr12 - sr6) - n*shift + tail
        return E_single
```

Listing 3: The single particle energy function. Note that the two different possitional states can be evaluated, the normal position, `self.x`, and optionally the trial position list `self.x_trial`.

1.c Virial pressure

The virial pressure can be computed using the same particle distances as derived using listing 1. Besides that, the pressure resulting from these interactions can be computed. For the pressure the current state is needed, including the computed list of particle distances. Just like the energy functions, from this state, the constants ε and σ are needed. The number density can be computed from the box size and the list with particle distances. The distances that are used already satisfy the minimal image convention and self interactions are removed. As there is no correction option included for the virial pressure equation, the decision is made to not cut off the particle interactions. In python, the code for the virial pressure is the following:

```
def pressure(self):
        T = self.T
        sig = self.sig
        eps = self.eps
        L = self.L
        V = L**3
        d_sq = self.d_sq
        sr2 = sig**2/d_sq
        sr6 = sr2*sr2*sr2
        sr12 = sr6*sr6
        P = co.k*T*d_sq.shape[0]/V - 24*eps*np.sum(sr6 -
        ↪  2*sr12)/(6*V)
        self.P = P
        return P
```

Listing 4: The virial pressure function.

It must be noted that the pressure calculation and the total energy computation both use summations over the Van der Waals and Pauli interaction. Computational efficiency could be increased by creating a single function which computes the $\left(\frac{\sigma}{r_{ij}}\right)^6$ and $\left(\frac{\sigma}{r_{ij}}\right)^{12}$ for both the total energy and pressure. However, then the functions of the observables would mix up a little, decreasing the readability of the code.

1.d Translation moves

For the Monte Carlo simulation, every iteration exist of a trial move, which is than accepted or rejected by assessing the change in energy. For this, the current state of the system is used. Besides the particle positions, the temperature is needed to evaluate the acceptance probabilities. The translation move function performs a relatively simple task: It chooses a random particle, applies a random displacement after which the energy of this single particle is tested in the original configuration and the trial configuration. The trial configuration is accepted using the following probability criterion:

$$\text{acc}(\text{old} \to \text{trial}) = \min\left(1,\ e^{-\frac{U_{\text{trial}} - U_{\text{old}}}{k_B T}}\right) \tag{4}$$

In other words, if the energy of the trial configuration is lower than that of the original configuration, the trial state will be accepted. If the trial energy is higher than that of the original configuration the acceptance probability decreases by Boltzmann statistics. This probability decreases the larger U_{trial} is compared to u_{old}.

```python
def translate(state):
        x = state.x
        N, D = x.shape
        T = state.T
        L = state.L
        max_step = state.max_step

        Ni = np.random.randint(0, high=N)
        trial_move = 2*max_step*np.random.rand(1, 3) - 1*max_step
        x_trial = np.copy(x)
        x_trial[Ni, :] = (x_trial[Ni, :] + trial_move) % L
        state.x_trial = x_trial

        U = state.singleEnergy(Ni)
        U_trial = state.singleEnergy(Ni, trial=True)
        dU = U_trial - U

        if dU < 0:
                state.x = state.x_trial
                state.accept = state.accept + 1
        else:
                p_acc = np.exp(-dU/(co.k*T))
                if p_acc > np.random.rand():
                        state.x = state.x_trial
                        state.accept = state.accept + 1
```

Listing 5: The translate function. Note that for the trial move, an explicit copy is made of the original configuration to be able to treat both configurations separately. The single particle energies for both configurations are computed using listing 3 and the trial configuration is accepted using Boltzmann statistics.

1.e Initial configuration setup

The initial configuration for the Monte Carlo method exists of two function in my code. The first function initialises the python Class that I build and the second function places the particles in their initial position. The inputs asked are the following: the Temperature in in K, the density in kg/m^3, the atomic mass of the particles in amu, the LJ energy constant ε in units of k_B and the LJ distance constant σ in Å.

The special input is the option `N_or_file`, which needs either the file location with the initial particle positions or the number of particles needed to model. This parameter is then passed to another function, the initial configuration function. Here, two options exists:

1. `N_or_file` is of string type. In that case the file location corresponding to the string is read and the particle positions are copied from these positions. The box size is then estimated from the particle positions and the density is recalculated as well. Besides that, the cut-off distance is set to half the box size by default.
2. `N_or_file` is of type integer. This results in calculating the box size for the inputted density and atomic mass and the amount of asked for particles. After that, N particles are placed in random positions within this box size. r_{cut} is again set to the default $L/2$.

Besides these initialisation functions, a last function exists where optional model changes are made. This function can be used to change the default cut-off distance as well as change the default values for the shift and tail corrections. These are, by default, set to 0, however can be set to correctly represent a

```python
class State:
        def __init__(self, T, rho, m_a, eps, sig, N_or_file):
                self.T = T
                self.rho = rho
                self.m_a = m_a
                self.eps = eps*co.k
                self.sig = sig*1e-10
                self.tail = 0
                self.shift = 0
                self.initialConfiguration(N_or_file)

        def initialConfiguration(self, N_or_file):
        if type(N_or_file) is str:
                data = pd.read_csv(N_or_file, delim_whitespace=True, header=None, skiprows=2)
                self.x = (np.array((data[1], data[2], data[3])).T)*1e-10
                self.L = np.round(np.max(np.abs(self.x*1e10)))*1e-10
                self.Rcut = self.L/2
                self.rho = self.m_a*self.x.shape[0]/(1000*co.N_A*self.L**3)

        elif type(N_or_file) is int:
                self.L = np.power(self.m_a*N_or_file/(co.N_A*1000*self.rho), 1/3)
                self.Rcut = self.L/2
                self.x = np.random.rand(N_or_file, 3)*self.L

        else:
                raise ValueError("N_or_file has to be the ammount of particles",
                "int, or the file location with an initial state")
```

Listing 6: These functions initialise the class.

```python
                def modelCorrections(self, Rcut=False, Tail=False, Shift=False):
                sig = self.sig
                eps = self.eps

                if Rcut is not False:
                Rcut = Rcut*1e-10
                if Rcut > self.L/2:
                print('Cutoff distance is more than half the box size, as',
                'remidy, Rcut is set to default, Rcut = L/2')
                else:
                self.Rcut = Rcut

                Rcut = self.Rcut
                if Tail is not False:
                tail_factor = (8*np.pi*N*eps*np.power(sig, 3))/(3*np.power(L, 3))
                tail_distances = (np.power(sig/Rcut, 9)/3) - np.power(sig/Rcut, 3)
                self.tail = tail_factor*tail_distances

                if Shift is not False:
                shift_factor = 4*eps
                shift_distances = np.power(sig/Rcut, 12) - np.power(sig/Rcut, 6)
                self.shift = shift_factor*shift_distances
```

Listing 7: The optional code to call the model correction terms. These correction terms are all collected into one function, however can be set to true or false individually.

truncated Lennard-Jones potential.

1.f Averaging for the results

Using the ergodicity postulate, the average of the observables over all configurations approaches the (true) time average of these observables. Therefore the observables are averaged over the configurations visited by the Monte Carlo algorithm. As only undergoing one trial move will not change the observables by much and computing the observables takes considerable computational effort, these observables are only collected every $2N$ trial moves. Still, some autocorrelation may still exist in the collected data. Therefore, using the standard deviation of the data is not the correct way to determine the uncertainty in the result. To have a better expression for the error in the collected data, firstly, the autocorrelation is computed and fitted to an exponential function with a unknown decay factor τ. The estimated uncertainty of the averaged value is then computed using the following equation;

$$u_{\text{uncertainty}} = \sqrt{2\tau \frac{\sum_i (u_i - \langle u \rangle)^2}{n}} \tag{5}$$

where u_i are all calculated values of an observable and $\langle u \rangle$ is the average of these values. n represents the total number of samples and τ the correlation decay factor. The code needed for the correct averaging and estimating the is presented below.

```
def statistics(s):
        N = s.shape[0]
        mean = s.mean()
        var = np.var(s)

        if var == 0.0:
                mean, error, tao, g = mean, 0, 0, 0

        else:
                sp = s - mean
                corr = np.zeros(N)
                corr[0] = 1
                for n in range(1, N):
                        corr[n] =
                        ↪ np.sum(sp[n:]*sp[:-n]/(var*N))
                g = np.argmax(corr < 0.1)
                t = np.arange(2*g)
                tao = opt.curve_fit(lambda t, b:
                ↪ np.exp(-t/b),  t,
                corr[:2*g], p0=(g))[0][0]
                error = np.sqrt(2*tao*s.var()/N)
        return (mean, error)
```

Listing 8: The code that averages the observables and computes their uncertainty while adjusting for autocorrelation.

1.g The Monte Carlo function, linking everything together

All the pieces of code are connected together by the Monte Carlo function. This function has the following input; the state used (which has to be the initiated class), the amount of times that the observables are sampled, the initial maximum step size and an optional setting to turn on or of the equilibrating of the initial configuration. Between the computing of consecutive observables, $2N$ trial steps are performed. The max step size for the trial move is also adjusted every $2N$ trials. This function is explained in listing 10. The current version of the Monte Carlo algorithm tracks three observables, the total energy in the system, the pressure of the system and the radial distribution function. The Monte Carlo function itself is presented below:

```
def monteCarlo(state, n, max_step_init, startup_eq=True):
        state.max_step = max_step_init*1e-10
        x = state.x
        L = state.L
        N, D = x.shape

        if startup_eq is True:
                for j in range(100):
                state.accept = 0
                for i in range(N):
                        translate(state)
                state.acceptance = state.accept/(N)
                state.newStepsize()

        # Now real measurement data is generated
        E_tot = np.zeros(n)
        P = np.zeros(n)
        rad_dis = np.zeros((1000, n))
        state.allDistances()
        r = np.histogram(np.sqrt(state.d_sq), bins=1000,
        ↪ range=(0, L/2))[1]
        r = r[:-1] + (r[1]-r[0])/2  # remove last bin edge
        ↪ and shift to center bins

        for i in range(n):
                state.accept = 0
                for j in range(2*N):
                        translate(state)
                state.acceptance = state.accept/(2*N)
                state.newStepsize()
                state.allDistances()
                E_tot[i] = state.totalEnergy()
                P[i] = state.pressure()
                n_r = np.histogram(np.sqrt(state.d_sq),
                ↪ bins=1000, range=(0, L/2))[0]
                rad_dis[:, i] = (L**3*n_r)/(N*(N-1)*4*np.pi*
                ↪ (r**2)*(r[1]-r[0]))
                state.progress = i/n
                update_progress(state.progress)

        state.progress = 1
        update_progress(state.progress)

        state.trial_moves = 2*N*n
        state.E_tot = E_tot
        state.P = P
        state.rad_dis = rad_dis
        state.r_bins = r

        E_tot = statistics(E_tot)
        P = statistics(P)
        rad_dis_m = np.zeros((rad_dis.shape[0], 2))
        for i in range(rad_dis.shape[0]):
                rad_dis_m[i, :] = statistics(rad_dis[i, :])
        return E_tot, P, rad_dis_m, r
```

Listing 9: The Monte Carlo function. The logic is as following, first an optional MC process to reach equilibrium is performed, after which the observables are introduced as empty arrays and then the MC algorithm is executed while storing the observables every $2N$ trial moves. After the MC process had run, the arrays containing the observables are stored in the class, but he average results and their uncertainties are returned immediately.

2 Retrieving results

2.a Assessing step size for liquid methane

Setting the optimal maximum step size, to stay in a acceptance ratio of 40 % to 50 %, by trial and error is prone to errors. Especially if these step sizes are determined when non-equilibrated configurations are used. To solve this problem, a function is build that updates the maximum step size according to the success rate of the last few trials. This function uses the acceptance rate, which is kept track of in the Monte Carlo function, listing 9. The function, as shown in listing 10, increases the maximum step size if the acceptance rate is too high, and reduces the maximum step size if the acceptance rate is too low.

```
def newStepsize(self):
        acceptance = self.acceptance
        max_step = self.max_step

        if acceptance <= 0.1:
        max_step *= 0.2
        elif acceptance < 0.4 and acceptance > 0.1:
        max_step *= 0.8
        elif acceptance > 0.5 and acceptance < 0.9:
        max_step *= 1.2
        elif acceptance >= 0.9:
        max_step *= 2

        if max_step > self.Rcut/10:
        max_step = self.Rcut/10
```

Listing 10: The function which optimises the step size.

To still answer this question in the assignment appropriately, the development of the maximum step size and the acceptance rate are tracked during the part of the algorithm that equilibrates the system. For 362 particles, it results in fig. 1. From this, I conclude that the advised max step size is quite close to the optimal maximum step size. During the main part of the Monte Carlo algorithm, the step size keeps being updated, however it stays close to 0.5 Å.

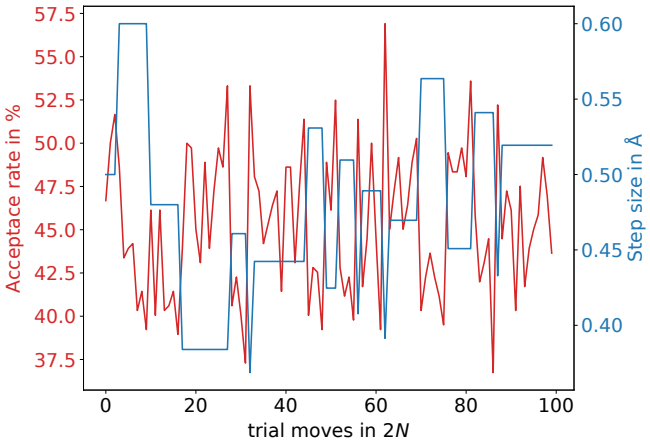

Figure 1: The development of the acceptance rate and the maximum step size during the equilibration phase for the liquid phase.

2.b Assessing step size for gaseous methane

For the gaseous state of methane, at 400 K and 9.68 kg/m^3, an interesting observation is made. The step size does not seem to influence the acceptance rate any more. This is because the density is so low, that the Lennard-Jones interactions are close to zero. A replaced particle will therefore still be likely to not interact with other particles by much. With the automatic maximum step size adjustment function, this behaviour can lead to exponential growth of the maximum step size. To keep some physical logic in the development of the state of the system, a maximum step size of $r_{cut}/10$ is used. The choice to make this limit dependant of r_{cut} instead of σ or L was arbitrary. This is just implemented to keep the maximum step size under an upper limit.

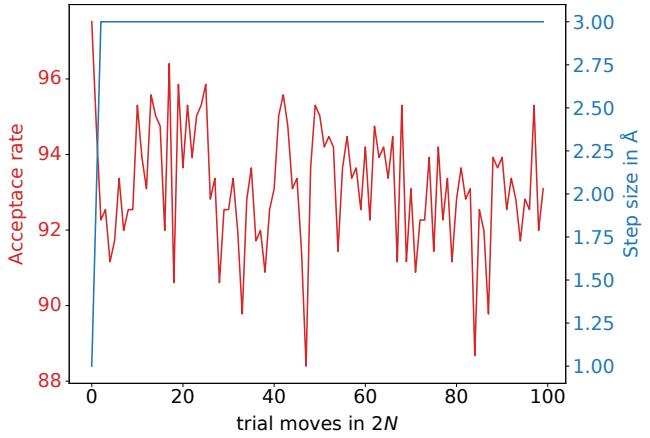

Figure 2: The development of the acceptance rate and the maximum step size during the equilibration phase for the gaseous phase. It is clear that the acceptance rate lies far above the 50 %, but the maximum step size is kept constant. This is because the density of the ensemble is so low that moving a particle to any place in the box results in a ≈ 90 % acceptance rate. Therefore, no max step size that result in a 40 % to 50 % acceptance rate exists.

2.c Checking if the method equilibrates correctly

To check if the code calibrates correctly, the model for 1000 particles is run twice, and the energy is plotted in fig. 3 and fig. 4. The first time, the equilibration phase is turned of, while the second computations is equilibrated first before simulating. Without equilibrating, the average result for the energy and the pressure are "out of control", reaching values of $E_{tot} = (3\pm3)\times10^4$ zJ and $P = (2\pm1)\times10^8$ bar. If the set is equilibrated first, the values are more logical, resulting in $E_{tot} = (-9.889\pm0.005)$ zJ and $P = (31\pm4)$ bar.

2.d Comparing different ensemble sizes

To compare the effects of the amount of particles on the pressure, the energy and the computation time, the model is ran for $N = 362$, 1000 and 3500 for three thermodynamically different cases. Case one, shown in table 1, was performed on a liquid state, 150 K at 358.4 kg/m^3. The other cases was in gaseous phase at 400 K at 9.68 kg/m^3 and its results are shown in table 2. The observables were sampled 1500 times, for each simulation, and sampled every $2N$ trial moves.

From these results it can be recognised that the pressure is more dependant on the number of particles and the error of this observable is larger than that in the potential energy of the ensemble. Besides that, the computational efficiency seems to be quite good, only taking 200 s of computation time for the case with 362 particles.

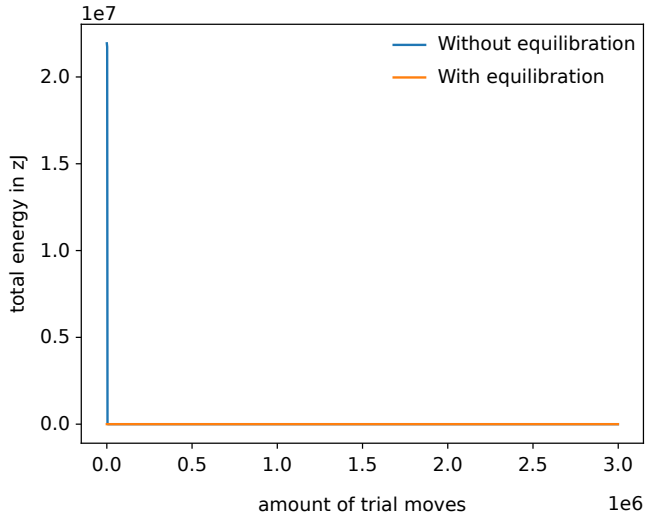

Figure 3: Zoomed out, no details are seen in the fluctuations. However, it is clear that initially the simulation without equilibration first has much larger energies, although these go down quickly. The simulation with initial equilibration clearly stays constant on this scale of energies. The results without equilibration seems to be vertical only, but it has a horizontal part as well, the blue line ended up behind the orange one.

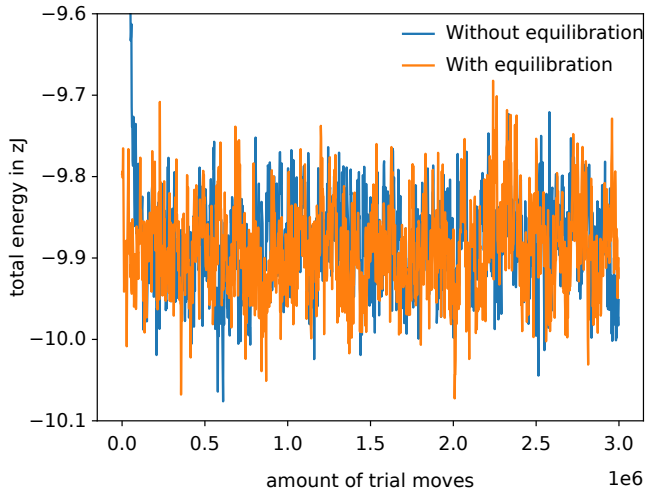

Figure 4: When the scale of the energy is zoomed in to discern the energy fluctuations, it is clear that except the initial trial moves, both models converge to a similar energy level they fluctuate around. However, the initial peak of the non-equilibrated influences the averages too much to be useful.

N	E per particle in zJ	P in bar	C-time in s
3500	−9.933±0.002	21±2	10 300
1000	−9.890±0.004	34±4	1020
362	−9.681±0.005	39±5	160

Table 1: The Liquid phase results. Each measurements consists of 1500 observables sampled every $2N$ trial moves. The cut-off distance is set to $L/2$ for the larger ensembles, for the ensemble with 362 particles a cut-off distance of 14 Å used. For these results, the shift and tail corrections are used. The computation time is recorded as well (C-time).

N	E per particle in zJ	P in bar	C-time in s
3500	−0.2392±0.0003	19.896±0.001	6800
1000	−0.2379±0.0004	19.9016±0.0006	1090
362	−0.2387±0.0007	19.90±0.01	220

Table 2: The Gaseous phase results. The cut-off distance is set to $L/2$ for the larger ensembles, for the ensemble with 362 particles a cut-off distance of 30 Å used. For these results, the shift and tail corrections are used. The computation time is recorded as well (C-time).

3 Results

3.a Isotherms of liquid and gaseous methane

The ultimate goal of this assignment is to be able to compute the classical thermodynamic state of methane: the relationship between T, ρ -or equivalently v- and P. The code computes the canonical ensemble, so ρ and T are set naturally, while the pressure is computed. To test the dependants of the pressure an isochore in both liquid and gaseous state is simulated. As the algorithm can take quite some time to run, only few points in the phase space are assessed. These isochores are presented in different graphs, fig. 5 and fig. 6, as liquid and gaseous methane scale quite differently with their temperature.

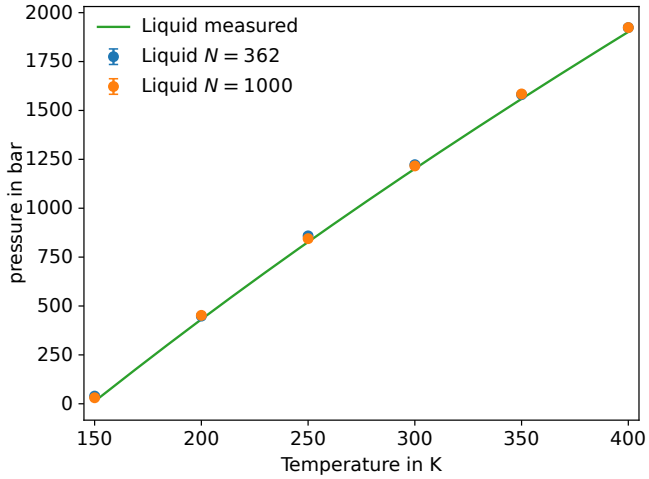

Figure 5: The isochore of methane at 358.4 kg/m^3 simulated using 362 and 1000 particles compared to measured data. The simulated values correspondent nicely with the measured data. Although the simulation using 1000 particles performs somewhat better than the 362 particles simulation, both seem to predict the development of pressure as function of the temperature quite accurately.

3.b The radial distribution function

To get further understanding of molecular simulations and the fundamental difference between the liquid and gaseous phase, the radial distribution function is computed as well. To investigate the behaviour of these distributions in detail, the simulations with 3500 particles from table 1 and table 2 are used. The cut-off distance of these simulations was $L/2$, resulting in long, tails. The resulting graphs, fig. 7, shows smooth behaviour. No discontinuities can be recognised in the radial distribution function. This indicates the correct implementation of the LJ potential and its corrections.

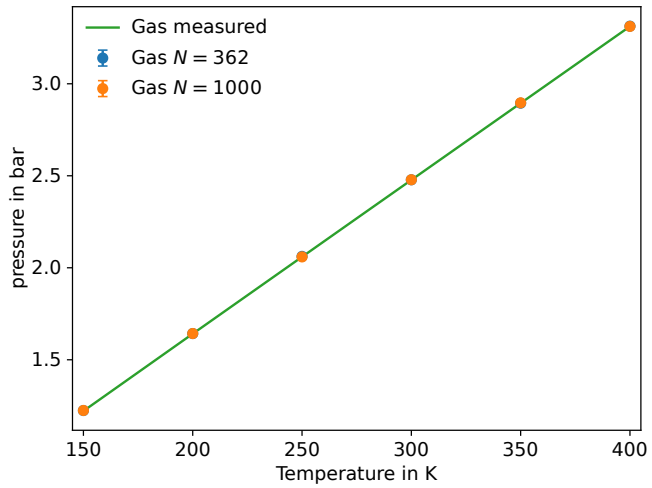

Figure 6: The isochore of methane at 1.6 kg/m^3 simulated using 362 and 1000 particles compared to measured data. The simulated values correspondent nicely with the measured data. For methane in its gaseouse phase, both simulations result in comparable, small, deviations from the measured values. However, the 1000 particles simulation results in significantly lower estimated error. Compared to fig. 5, the isochore in gaseous phase is clearly linear, this is in accordance with ideal gas behaviour.

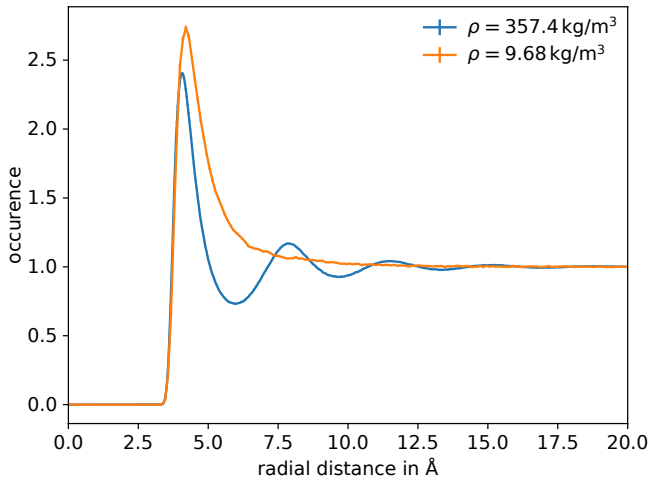

Figure 7: The radial distribution function of liquid and gaseous methane. As the ensemble exists of 3500 particles and 10.5×10^6 Monte Carlo trials are executed, the local errors in the probability density function are small.

A Maybe an appendix