Worksheet 3

Jens Jakschik, Fabio Oelschläger

February 1, 2021

Source code via Github
https://github.com/bliepp/Simulation-Methods-in-Physics-Exercises

Exercise 3: Saving and restarting the simulation

Goal of this exercise was to extend the simulation program given for this task so that it is possible to restart the simulation at the point where the previous simulation stopped. This is highly useful, as simulations can take excessive amounts of time depending on the task. This makes it highly attractive to reuse previous simulation results instead of repeating the same steps. The function to save a state file, which can be used to resume the simulation, was already included in the program, but was not fully implemented. The program has a parser which loads and saves a file depending on the parameter given during the starting of the program. For this state file to fully save the state of the system at the end of the simulation, it needs to save multiple parameters.

```
if args.cpt:
    state = {
        'dt': DT, # saving for convenience
        'sampling_stride': SAMPLING_STRIDE, # saving for convenience
        'energies': energies,
        'positions': positions,
        'pressures': pressures,
        'temperatures': temperatures,
        'rdfs': rdfs,
        'last_v': sim.v
        }
    write_checkpoint(state, args.cpt, overwrite=True)
```

With this extension, not only the energies are saved, but also the positions, pressures, temperatures, rdfs and the last velocities. This allows for a complete continuation of the simulation. The information saved here is actually more than what is needed to simply resume the simulation, but in later parts it is valuable to save the calculated observables, which is why it is already implemented here. The time step and checkpoint interval are saved here, too, to make the evaluation a bit more convenient.

Exercise 4: Simple Observations

For this part of the exercise, the existing code should be expanded, so that in addition to the existing code, additional observables are calculated and returned when the simulation finishes. The returning of the observables was already implemented in the previous exercise, which is why here we will only showcase the functions with which the different observables were calculated. First, the kinetic and potential energy components were to be calculated and returned, which was done with the following code. For the kinetic energy we simply used the well known formula for this, while for the potential we utilized potential matrix which was utilized earlier in the code.

```
def kinetic_energy(self):
    return 0.5*sum(sum(self.v*self.v))
def energy(self):
    self.energies()
    e_kin = kinetic_energy()
    e_pot = sum(itertools.chain(*self.e_pot_ij_matrix))
    return e_kin,e_pot
```

The next observable to be calculated was the temperature of the system. For this, the following equation was used:

$$\frac{1}{2}k_{\mathrm{B}}T_{\mathrm{m}} = \frac{E_{\mathrm{kin}}}{D \cdot N} \tag{1}$$

This means, that to calculate the temperature, only the number of particles N, the kinetic energy and the degrees of freedom D are necessary. This was done with the following function. As we are in reduced units, k_{B} can be assumed to be one.

```
def temperature(self):
    k=1
    e_kin=self.kinetic_energy()
    Temp = 2/k *e_kin/(self.n*self.n_dims)
        return Temp
```

Finally, the pressure inside the system was to be calculated. Here, again the kinetic energy is needed. This means, that the kinetic energy function is called a total of three times, which is sub-optimal. But, due to the fact that the different observables are calculated separately from each other in no specific order, this is the easiest course of action. To actually calculate the pressure, the force and the position of the particles is needed. Here, just like with the temperature, the provided formula was used.

$$P = \frac{1}{2A}\left(\sum_{i=0}^{N} m\underline{v}^2 + \sum_{i=0, j<i}^{N} \underline{F}_{\mathrm{ij}} \cdot \underline{r}_{\mathrm{ij}}\right) \tag{2}$$

```
def pressure(self):
    e_kin = self.kinetic_energy()

    force_part = 0
    for i in range(1, self.n):
        for j in range(i):
            f_ij = self.f_ij_matrix[i, j]
            r_ij = self.r_ij_matrix[i, j]
            force_part += np.dot(f_ij, r_ij)

    A = np.prod(self.box)
    return (e_kin + 0.5*force_part)/A
```

Exercise 5: Equilibration

Next, the simulation was run for 1000 time units. The observables were saved and plotted in the following figures. In addition to this, an additional function to calculate the running averages of the observables was to be implemented. Running averages are used to smooth data. The function calculates a running average with O being an array containing the time series of the observable and M being the window over which the running average is calculated.

```
def running_average(O, M):
    for i in range(len(O)):
        if (i < M) or (i > len(O)-M-1):
            temp_sum = float("nan")
        else:
            temp_sum = np.sum(O[i-M:i+M+1]) / (2*M+1)
        yield temp_sum
```

The running averages of the observables were calculated with 49 total particles and a window size of 10 and 100, the result of which can be seen below. These plots showcase how important equilibration is for an accurate simulation, as the starting values do not represent the intended system at all. As the particles of the MD simulation start on a lattice, the initial values of the observables start at very different values from what they would have in an equilibrated state. The averages first drastically change in the first time units, until they stabilize around an average, meaning the system has equilibrated. Due to figure 1 we are confident to say, that the system is fully equilibrated after about 200 time units. This statement is supported by figure 2.

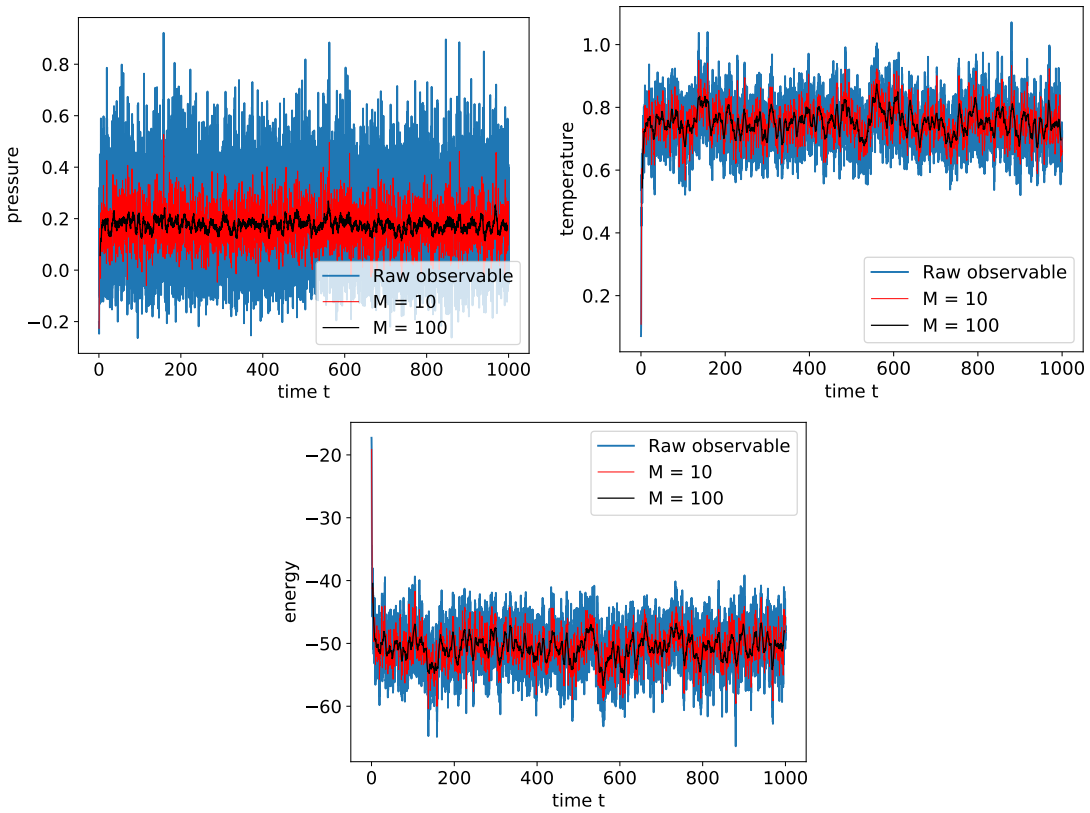

Figure 1: Running Averages of Pressure, Temperature and kinetic energy with different window sizes for 49 particles.

As it can be seen in figures 1 and 2 the observables drift towards a specific value. This is because initializing a system on a lattice forces the system to obtain a specific potential energy. This very high induced potential energy gets converted into kinetic energy (and therefore increases the temperature and pressure of the system). This happens in both ways until the overall energy components do not change much in average.

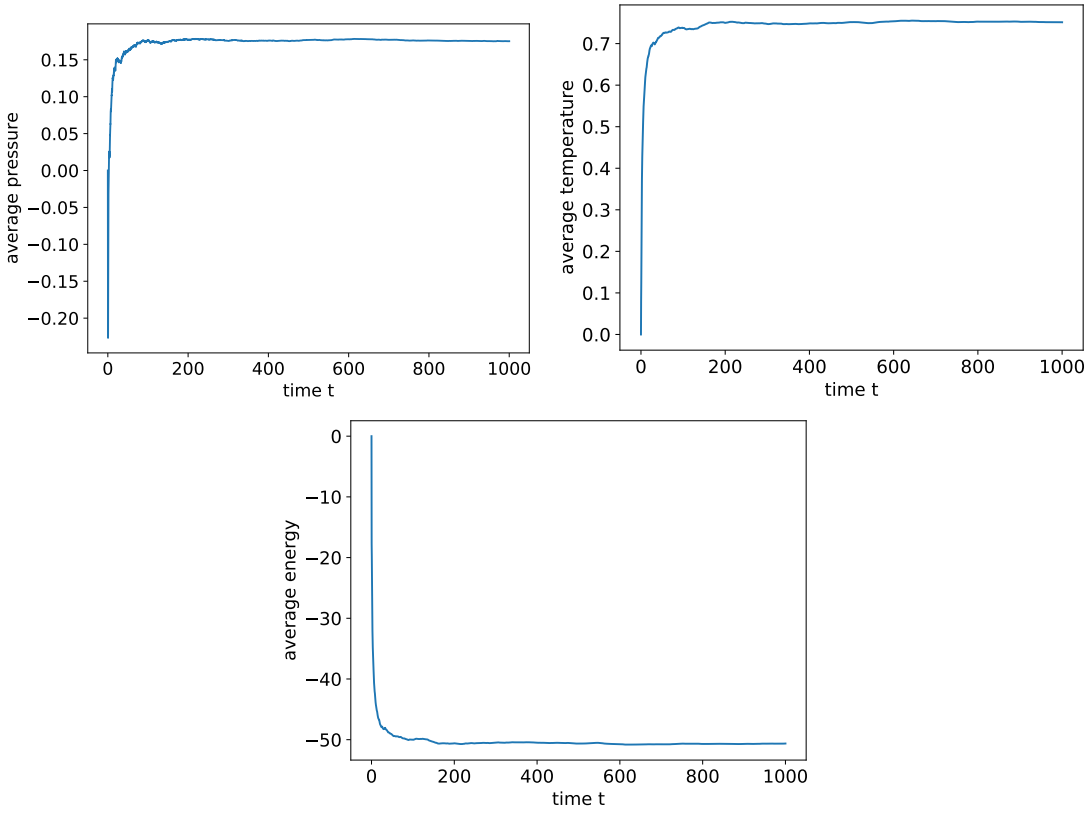

Figure 2: Total averages of Pressure, Temperature and kinetic energy after a given time.

Exercise 6: Molecular Dynamics at a Desired Temperature

If a simulation at constant temperature is needed, one simple way toachieve this is to rescale all velocities to fit the desired temperature. The factor needed to rescale can be derived from the equipartition theorem in equation (1). Assuming that every velocity is scaled by the same factor $v = v_0 \cdot a$ the kinetic energy needed becomes $E_{\text{kin}} = a^2 E_{\text{kin},0}$ where x_0 defines the variable x before rescaling. The equipartition theorem therefore becomes equation (3). This results in a factor a as shown in equation (4).

$$\frac{1}{2} k_{\text{B}} T = \frac{E_{\text{kin}}}{N \cdot D} = \frac{a^2 E_{\text{kin},0}}{N \cdot D} \tag{3}$$

$$a = \sqrt{\frac{N \cdot D \cdot k_{\text{B}} T}{2 E_{\text{kin},0}}} \tag{4}$$

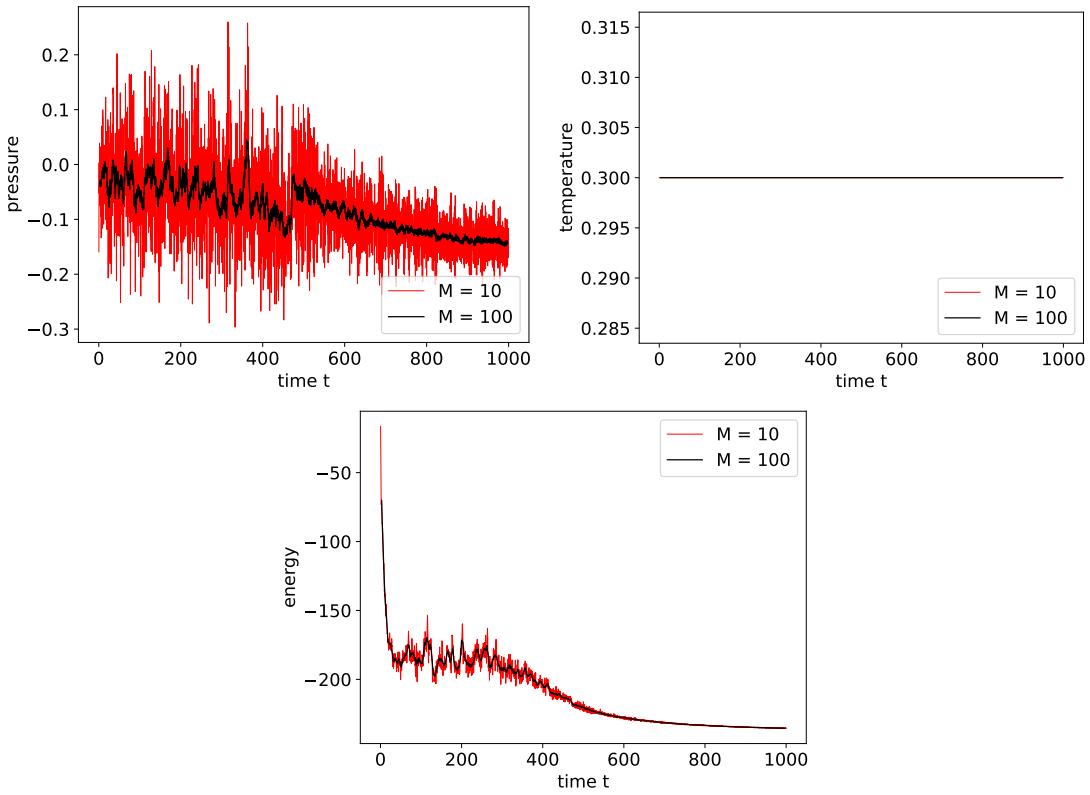

Figure 3: Simulation of the system at a given temperature $T = 0.3$ using the velocity rescaling model.

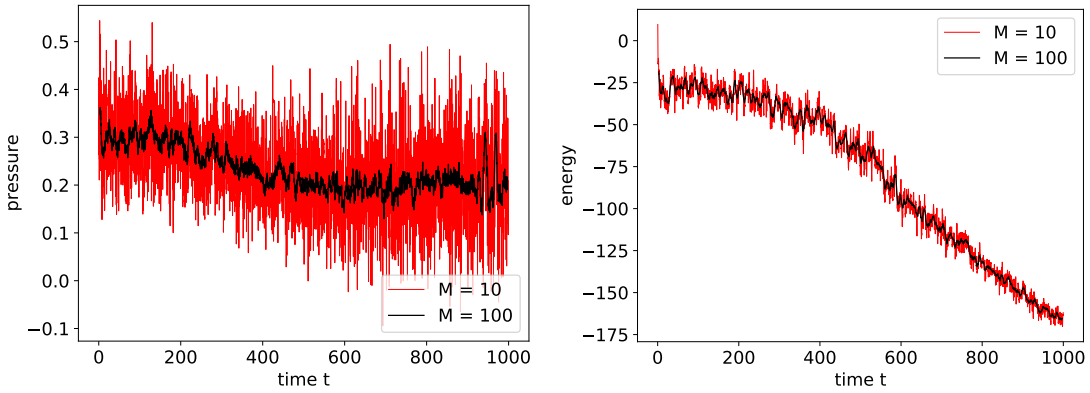

Figure 4: Simulation of the system at a given temperature $T = 1.0$ using the velocity rescaling model.

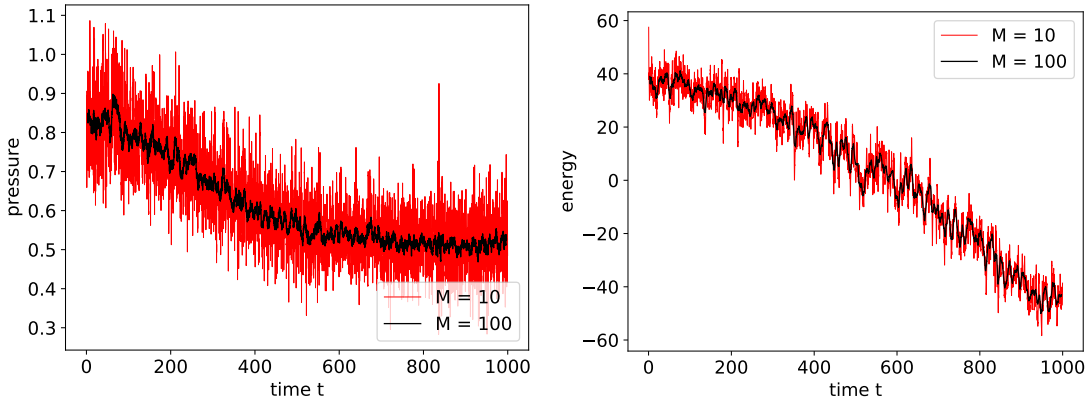

Figure 5: Simulation of the system at a given temperature $T = 2.0$ using the velocity rescaling model.

Exercise 7: Setting up and Warming up the System

For this exercise, the existing simulation was to be extended, so that the particles are placed randomly inside the box and it performs a so-called warmup before the actual simulation. For this, a maximum force is introduced during the warmup. This maximum force limits the interaction strength during the warmup and it can be determined that the warmup is finished by checking if the force is still being limited or not. When the forces do not need to be limited anymore, the system has warmed up and there are no more overlapping particles. Together with the rescaling of the velocities performed in the previous exercise, the simulation can be improved.

To be able to activate or deactivate the force capping at will, it was introduced as an additional argument in the parser, as well as an optional argument in the initial method. It is initialized as FMAX = None, so that the if statement where the force is capped is only used when a maximum Force is given when the simulation is started. The force is capped by extending the function that calculated the forces acting on the molecules so that all forces over FMAX are set to FMAX.

```
if self.FMAX and self.warmup:
    capped = []
    for i in range(0,len(self.f)):
        abs_ = scipy.linalg.norm(self.f)
        capped.append(abs_ > self.FMAX)
        if capped[-1]:
            self.f[i] = self.f[i]/abs_ * self.FMAX
    self.warmup = any(capped)
```

In addition to this, the simulation was changed so that the particles are not placed on a lattice, but instead randomly throughout the box, similar to the velocities which were already randomly generated.

```
if not args.cpt or not os.path.exists(args.cpt):
        logging.info("Starting from scratch.")
        #random particle positions
        x = (BOX * np.random.random((N_PART, DIM))).T
        # random particle velocities
        v = 0.5*(2.0 * np.random.random((DIM, N_PART)) - 1.0)
```

The maximum force doesn't stay constant however. With every step of the simulation, the maximum force is increased by 10 percent, to avoid the possibility of the warmup never finishing, because it is possible that the average force is larger than the initial maximum force. Before the warmup of the system is finished, the recorded data from the simulation is not physical, as there is an arbitrary cap on the forces acting between the particles. Due to this, the saving of the simulated data is changed, that the simulation only starts saving data once the warmup has finished, meaning that none of the forces between the particles have to capped anymore. For this, a while loop was included before the data is saved, which stays active until the warmup has finished. Some print values were included so that it is easy to see whether the warmup is working or not.

```
print("Beginning Warmup")
counter = 0
while sim.warmup and args.FMAX:
    sim.propagate()
    if counter % SAMPLING_STRIDE == 0:
        sim.FMAX = sim.FMAX * 1.1
    counter += 1
print("Warmup finished")
```

The warmup attribute of the simulation class is a parameter used to toggle between warmup mode and normal simulation mode. The results of these extensions can be seen in the following figures. There it can be seen that the system is equilibrated from the start, meaning the warmup of the system was performed successfully.

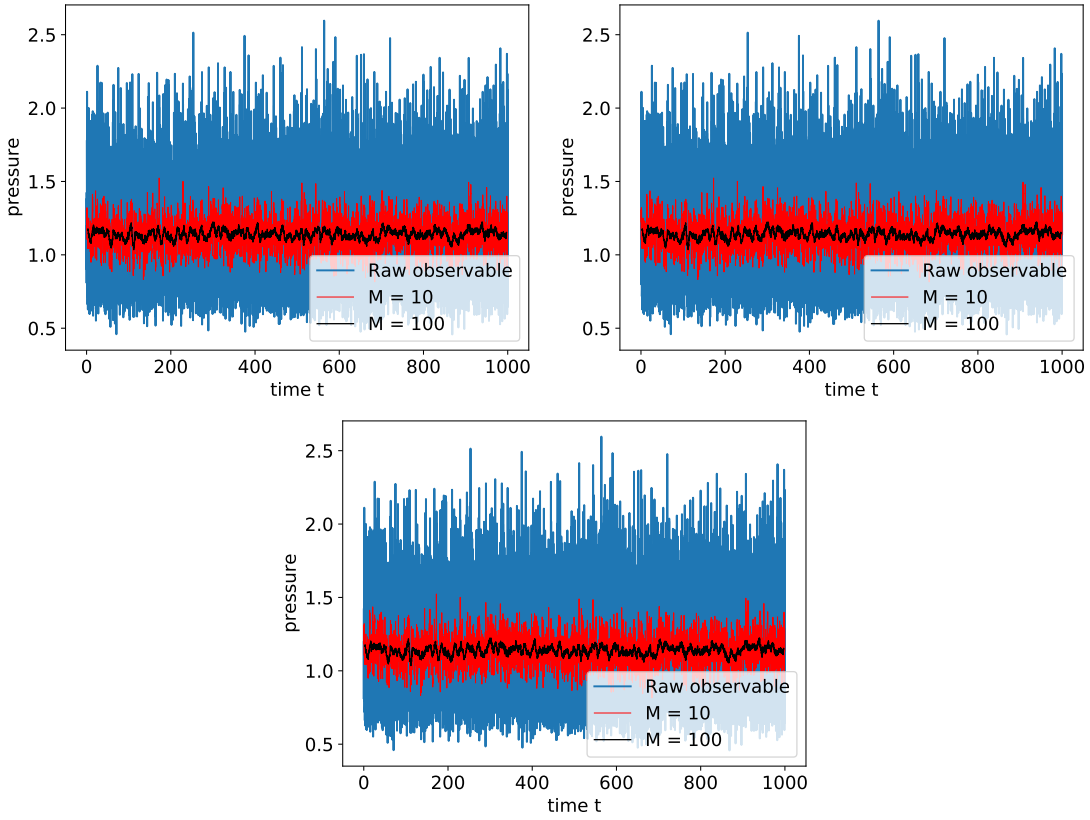

Figure 6: Different observables of the simulated system with a warmup done previously to saving the data

Exercise 10: Tail correction in the Pressure Calculation

While a cutoff range for the LJ potential saves computing time, it also introduces inaccuracies into the simulation. Therefore, the task in this exercise is to calculate the tail correction for the pressure in a system where the LJ potential determined the interaction between the particles. The LJ interaction potential is given by:

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{5}$$

To calculate the contribution of particles above the cutoff range, a tail correction is used, which is based on the assumption that the radial distribution function becomes 1 for distances larger than the cutoff range r_c. The radial distribution function is given by:

$$g(r) = \frac{1}{\rho 4\pi \underline{r}^2 \mathrm{d}r} \sum_{ij} \left\langle \delta(r - |\underline{r}_{ij}|) \right\rangle \tag{6}$$

To calculate the tail correction, we first have to establish the average potential energy of an atom in the system. This is given by:

$$u_i = \frac{1}{2}\int_0^\infty \mathrm{d}r\, 4\pi r^2 \rho(r) u(r). \tag{7}$$

Here, $\rho(r)$ stands for the average density around the atom i, and $u(r)$ for the LJ potential which determines the interaction between the atoms. The tail correction is responsible for interactions above the cutoff range r_c, which is why only the average potential energy above that cutoff range needs to be calculated.

$$u_\mathrm{i,tail} = \frac{1}{2}\int_{r_\mathrm{c}}^\infty \mathrm{d}r\, 4\pi r^2 \rho(r) u(r) \tag{8}$$

An additional assumption that has to be taken for this calculation is for the density $\rho(r)$. Here, we assume the density to be constant above the cutoff range, which is a viable assumption for most systems with a high degree of freedom and enough particles. This further simplifies the calculation to:

$$u_\mathrm{i,tail} = 2\pi\rho \int_{r_\mathrm{c}}^\infty \mathrm{d}r\, r^2 u(r). \tag{9}$$

With this, the tail correction for a LJ system can be calculated. Inserting the formula for the LJ potential results in:

$$\begin{aligned} u_\mathrm{i,tail} &= 8\pi\rho\epsilon \int_{r_\mathrm{c}}^\infty \mathrm{d}r\, r^2 \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] && (10)\\ &= 8\pi\rho\epsilon \int_{r_\mathrm{c}}^\infty \mathrm{d}r\, \left[\left(\frac{\sigma^{12}}{r^{10}}\right) - \left(\frac{\sigma^{6}}{r^{4}}\right)\right] && (11)\\ &= 8\pi\rho\epsilon \left[\frac{1}{9}\left(\frac{\sigma^{12}}{r^{9}}\right) - \frac{1}{3}\left(\frac{\sigma^{6}}{r^{3}}\right)\right]_{r_\mathrm{c}}^{\infty} && (12)\\ &= \frac{8}{3}\pi\rho\epsilon\sigma^3 \left[\frac{1}{3}\left(\frac{\sigma}{r_\mathrm{c}}\right)^{9} - \left(\frac{\sigma}{r_\mathrm{c}}\right)^{3}\right] && (13) \end{aligned}$$

The tail correction for the pressure is calculated similarly. Here, instead of just integrating over the potential however, the tail correction is determined by using the contribution of the potential towards the pressure in the system. This is given by:

$$P = \sum_\mathrm{ij}^N \underline{F}_\mathrm{ij} \cdot \underline{r}_\mathrm{ij} \tag{14}$$

Here, the force resulting from the LJ potential is needed, which was calculated from the previous exercise sheet. This results in the following calculation.

$$\Delta P_{\text{tail}} = \frac{1}{2} 4\pi\rho^2 \int_{r_c}^{\infty} \mathrm{d}r\, r^2 (\underline{r} \cdot \underline{F}(r)) \tag{15}$$

$$= 2\pi\rho^2 \int_{r_c}^{\infty} \mathrm{d}r\, r^2 \cdot 8\epsilon \left[2\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{16}$$

$$= 16\pi\rho^2\epsilon \int_{r_c}^{\infty} \mathrm{d}r \left[2\left(\frac{\sigma^{12}}{r^{10}}\right) - \left(\frac{\sigma^{6}}{r^{4}}\right) \right] \tag{17}$$

$$= \frac{16}{3}\pi\rho^2\epsilon\sigma^3 \left[\frac{2}{3}\left(\frac{\sigma}{r_c}\right)^{9} - \left(\frac{\sigma}{r_c}\right)^{3} \right] \tag{18}$$

Some steps were skipped here, as they are equivalent to the calculations above, which were done a little more in depth.