TIF320 – Assignment 3

Computational Materials and Molecules

–

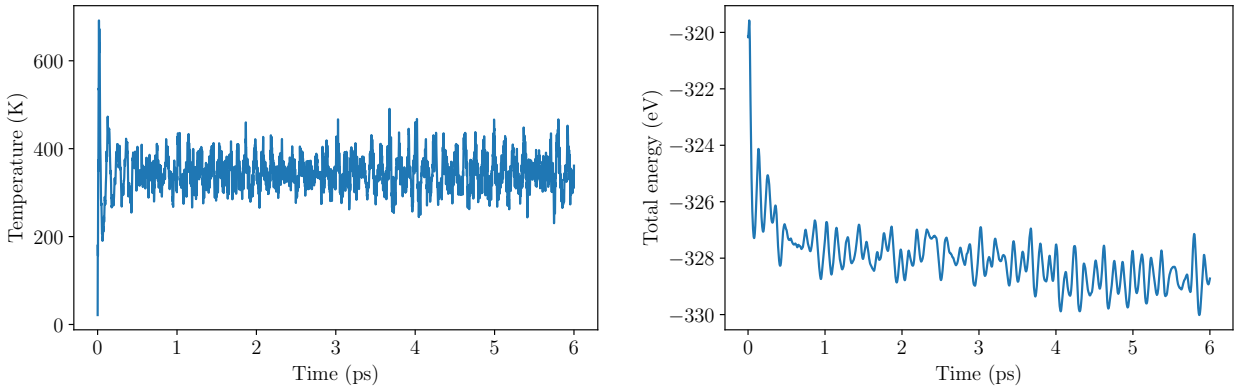
Na^+ solvation in water

Nico Guth (nicog) and Erik Levin (eriklev)

February 17, 2023

Task 1: Running the calculations

Given is a simulation of water using ab initio molecular dynamics (AIMD) that includes a thermostat. The evolution of the temperature and energy in this simulation is shown in fig. 1. The temperature reaches equilibrium after about 0.5 ps. However, the energy keeps dropping until about 4 ps into the simulation. This implies that, only then, the thermostat could be disabled and thermodynamic equilibrium has been reached. The noise in both the temperature and the energy can be explained by the finite simulation cell size.



(a) **(b)**

Figure 1: Temperature (a) and energy (b) trajectory of the given water AIMD simulation.

The last snapshot of this simulation was extracted, and a Na ion was placed inside. The modified simulation cell is visualized in fig. 2 and is the starting snapshot of the AIMD simulation performed by us.

Below follows the answers to the subquestions in task 1.

Task 1.1: Care should be taken when placing the ion, but why? What would happen if you placed it too close to a water molecule?

Placing the Na ion too close to a water molecule, would risk introducing repulsive forces into the system, causing it to behave unphysically during the simulation. For example, if the electronic

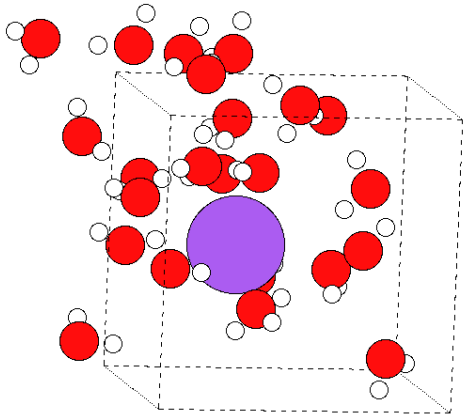


Figure 2: Starting point of the sodium solvation simulation.

density of the ion and the molecules were overlapping and then released, the water molecule would probably be shot away from the ion and then the simulation would be of another system than the one we want. Another possibility is that it would introduce a bond that is too strong and would create a NaO molecule, which is also not what we want to simulate.

Task 1.2: What is an appropriate timestep for MD simulations for this system?

We have chosen a timestep of 0.5 fs because all particles except the one Na ion are H_2O molecules and according to the lecture water has an appropriate timestep of approximately 0.5 fs. The reason for that is the rule of thumb of 20-40 timesteps in one period of the fastest oscillation in the simulated system. In water, the O−H stretching is the fastest characteristic oscillation, with a frequency of approximately 100 THz. To have 20 timesteps in one period of this oscillation, a timestep of 0.5 fs is needed. Since Na is a heavier element than both H and O, it has much slower oscillation frequencies. Therefore, a shorter timestep is not needed to capture the correct dynamics.

Task 1.3: Very briefly explain how the Nosé-Hoover thermostat keeps the temperature constant in an MD simulation.

The Nosé-Hoover thermostat is a so-called global thermostat that ensures that the total kinetic energy is distributed correctly. Global thermostats modify the Hamiltonian such that the system converges to the correct energy without changing the individual particles specifically. The main idea of Nosé-Hoover is the introduction of the heat bath into the Hamiltonian, therefore into the system itself. This is done by introducing an imaginary particle to represent the bath and with a coordinate s and a mass variable Q, which has to be chosen carefully, into the Hamiltonian

$$H_{\text{Nose}} = \sum_i^N \frac{\mathbf{p}_i}{2m_i s^2} + U(\mathbf{r}^n) + \frac{{\mathrm{p_s}}^2}{2Q} + L\frac{\ln s}{\beta} \tag{1}$$

with

$$\mathbf{p}_i = m_i s^2 \dot{\mathbf{r}}_i \quad \text{and} \quad p_s = Q\dot{s}\,. \tag{2}$$

Task 1.4: The Nosé-Hoover thermostat is not actually ergodic. Explain what the term "ergodic" means in the context of molecular dynamics and why it's an important concept.

A physical system is ergodic when all possible microstates of the same energy will be visited if the system evolves long enough in time. The ergodic hypothesis then says that the time averages of the thermodynamic system equal the ensemble average. Since a common goal of a molecular dynamics simulation is to find average quantities of the system (temperature, pressure, compressibility,...), it is important for a simulation that the system is ergodic.

Task 1.5: Despite this, the Nosé-Hoover thermostat is considered to be an excellent choice for simulating systems at finite temperature. How is the ergodicity problem solved?

An improvment of the Nosé-Hoover thermostat to solve the ergodicity problem can be achieved by the use of so-called Nosé-Hoover chains. The idea is to introduce more thermostats to the thermostat itself, or in other words, a chain of thermostats. This now increases the ergodicity in the system and the Hamiltonian can be described as

$$H_{NHC} = \sum_i \frac{\mathbf{p}_i}{2m_i} + U(\mathbf{r}^n) + \sum_j \frac{q_j \xi_i^2}{2} + g k_\mathrm{B} T s_1 + \sum_{j=2}^{M} k_\mathrm{B} T s_j\,. \tag{3}$$

Task 1.6: DFT gives the energy as a function of the density, but we need forces to do MD. How do we calculate forces within the framework of DFT?

To calculate forces within the DFT framework, the fact that DFT is a ground state theory is used. The electronic density is calculated using DFT in each timestep. Having obtained the ground state energy E_0, the use of the Hellmann-Feynman theorem is possible. The theorem states that the force to any parameter describing the system, such as the position $\mathbf{R}_n$ of the nucleus, can always be written as

$$\mathbf{F} = -\frac{\partial E_0}{\partial \mathbf{R}_n}\,. \tag{4}$$

This enables us to calculate the forces on the nuclei, which are then used in the molecular dynamics simulation.

Task 1.7: What else could be used than DFT for force calculations? Give an example! What are some of the advantages/disadvantages compared to DFT?

Instead of using DFT in each time step, a force field method could be used to calculate the forces in the molecular dynamics simulation. Here, the potential energy around nuclei is approximated in some form. Common examples are the Coulomb interaction or the Lennard-Jones potential that can be used. Then the forces can be calculated again as the gradient of the potential. This is cheaper computationally, but less accurate and applicable only on larger systems compared to the DFT. Therefore, force field methods can be a good choice if large systems with hundred thousand molecules need to be simulated, due to the decreased cost and maybe the accuracy of DFT is not needed in those cases to extract the wanted properties.

Task 2: Analysis of results

Task 2.1 Algorithm explaination

In order to compute the RDF, the first step is to acquire the distances from the sodium ion to each oxygen atom. The simplification of only considering the oxygen atoms, not the hydrogen atoms is done so that the RDF to the molecule is calculated and not to all atoms. These distances need to be binned into a histogram. Therefore, we first need to set the properties of the histogram, i.e. its range and number of bins. The bin centers are used in the following as r.

```
nbins = 200
rmin = 0.
rmax = 10.

bin_edges = np.linspace(rmin,rmax,nbins)
bin_width = np.diff(bin_edges)[0]
bin_centers = bin_edges[:-1]+bin_width/2
```

This gives us a granularity of 0.05 Å. We can now obtain the positions for each relevant atom by using the following code.

```
    elements = atoms.get_atomic_numbers()
    idx_Na = np.where(elements==11)[0]
    idxs_O = np.where(elements==8)

    pos = atoms.get_positions(wrap=True)
    pos_Na = pos[idx_Na][0]
    pos_O = pos[idxs_O]
```

Having acquired the positions for the given snapshot, we now make a copy of this cell 26 times around it. The argument for this and not using the minimum image picture was to enable greater ranges of radius comparisons, but an alternative solution would have been using the minimum image convention and placing the cell centered around the sodium ion. Having copied the cell, we now calculate the distances to all atoms and compute the distance histogram.

```
    # implement periodic boundary conditions by copying the O positions 26
    times around the main cell
    cell = atoms.get_cell()
    pbc = atoms.get_pbc()
    pos_all_O = [wrap_positions(pos_O,cell,pbc,center=[0.5+ix,0.5+iy,0.5+iz])
                 for ix in range(-1,2)
                 for iy in range(-1,2)
                 for iz in range(-1,2)]

    pos_all_O = np.concatenate(pos_all_O, axis=0)

    dists = (pos_Na[0]-pos_all_O[:,0])**2
    dists += (pos_Na[1]-pos_all_O[:,1])**2
    dists += (pos_Na[2]-pos_all_O[:,2])**2
    dists = np.sqrt(dists)

    bin_counts = np.histogram(dists, bin_edges)[0]
```

Having obtained the distances, we can now calculate the RDF

$$g(r) = \frac{\mathrm{dn_r}}{4\pi r^2 \mathrm{d}r} \frac{1}{\rho} \tag{5}$$

which gives us the following code,

```
    # calculate the radial distribution function
    rho = bin_counts.sum()/(rmax**3*np.pi*4/3)
    rho_local = bin_counts/(4*np.pi*bin_width*bin_centers**2)
    g = rho_local/rho
```

The steps above are then repeated for each timestep and the average over all timesteps gives the final RDF. As the coordination number for the first solvation shell is of special interest, the first minimum of the total RDF is searched for and the integration from the start to the first minimum yields the coordination number.

$$\text{Coordination number} = 4\pi\rho \int_{r_0}^{r_{1\text{st min}}} r^2 g(r) \mathrm{d}r \tag{6}$$

```
    first_minimum_idx = argrelextrema(g, np.less, order=4)[0][0]
    first_minimum = bin_centers[first_minimum_idx]
    corr_number = np.sum(g[:first_minimum_idx])*bin_width

    corr_number = 4*np.pi*rho*np.trapz(bin_centers[:first_minimum_idx]**2*g[:
    first_minimum_idx], bin_centers[:first_minimum_idx])
```

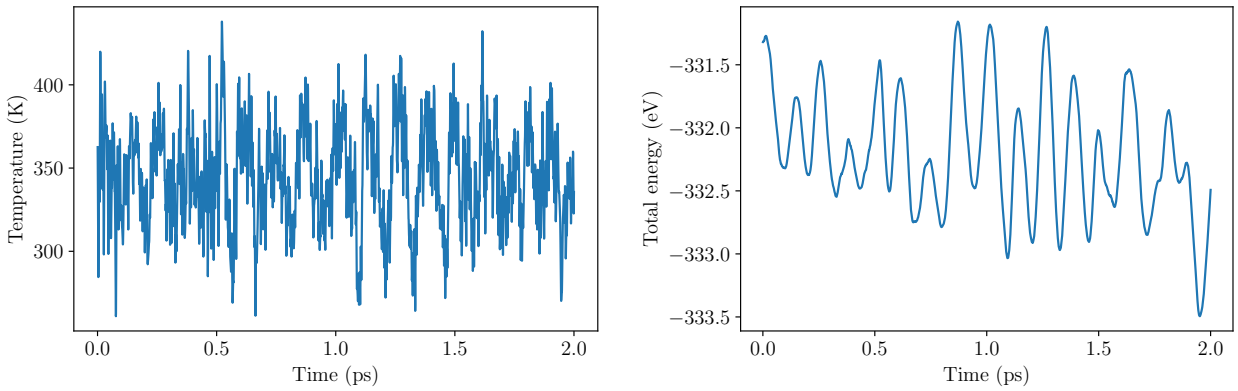
The results of the RDF calculation are show in fig. 6 for both our simulation and the given simulation.

Task 2.2: Was the 2 ps simulation enough to establish thermal equilibrium?

In fig. 3, the energy and temperature trajectory of our simulation is shown. Thermal equilibrium is reached almost immediately, as the temperature stays almost constant (with expected

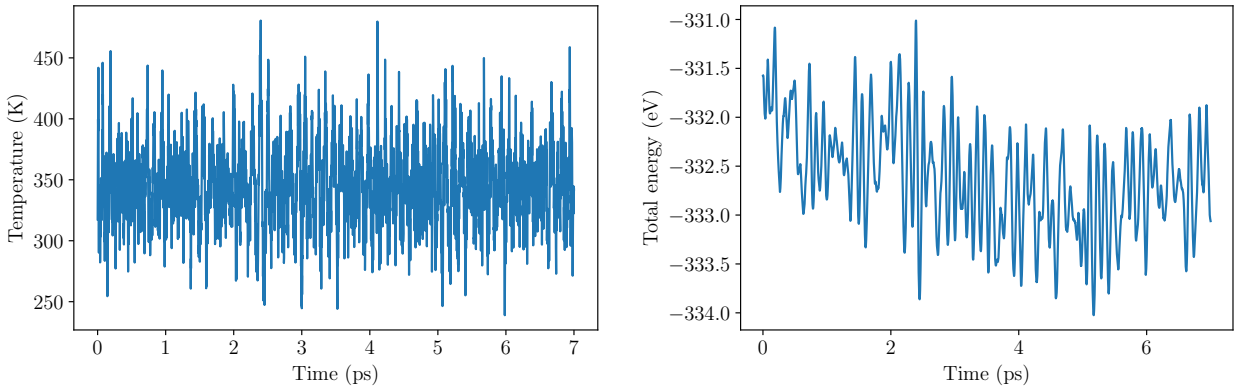
fluctuations). However, the system does not reach thermodynamical equilibrium as the energy is continuing to drop.

In fig. 4, the energy and temperature trajectory of the given simulation is shown. It is unclear whether an equilibrium was reached in this simulation, as the energy still changes and no clearly constant average has been reached.



(a) **(b)**

Figure 3: Temperature (a) and energy (b) trajectory of the performed AIMD simulation for 2 ps of the sodium solution.



(a) **(b)**

Figure 4: Temperature (a) and energy (b) trajectory of the given AIMD simulation of the sodium solution.

Task 2.3: The water molecules are oriented in a certain manner, both around the ion and far away from it. Explain why!

The orientation of the water molecules comes from the introduced positive charge by the sodium ion. One example snapshot of the performed simulation is shown in fig. 5. The asymmetry of the water molecule produces a slight dipole with a weak positive and weak negative side. As such, the water molecules in the first layer will arrange themselves with the weak negative

side, i.e. the oxygen atom, turned towards the sodium atom and the positive side, i.e. the hydrogen atoms, turned away from the sodium atom. The system outside the first solvation shell is also affected and there is some spacing between the first solvation shell and the second. Additionally, the water molecules are arranged in such a way that the oxygen atoms are closer to the hydrogen atoms than to other oxygen atoms.

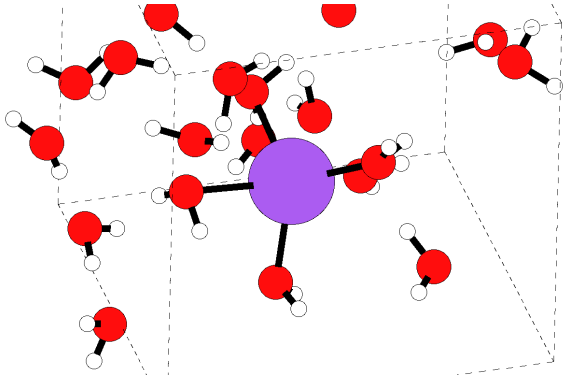


Figure 5: Solvation shell for the final snapshot of our simulation. Drawn are also the bonds between atoms. Observe the spacing between the first four bound molecules and the rest. Also, observe the overall trend of this shell with the charge alignment where the two hydrogen atoms are pointed away from the sodium ion.

Task 2.4: Was the short simulation enough to capture the solvation?

The solvation of sodium means that the individual sodium ions get separated, and a hydration shell is formed around those ions. The fact that there is a clear peak in fig. 6 shows that the solvation shell has formed. From this perspective, the 2 ps were enough to capture the solvation. However, the difference to the experimental value of 5 suggests that the solvation shell has not formed completely yet.

Task 2.5: Do the simulation results agree well with the experimental value of 5? What does this number tell us about the solvated ion?

To quantify the solvation, the so-called coordination number or hydration number is calculated through eq. (6). This resulted in a hydration number of 3.98 for our simulation of 2 ps and 4.65 for the equilibrated part (last 3 ps) of the given simulation. That means that on average, the first solvation shell consists of 4 water molecules. The experimental value of the hydration number is however 5. Therefore, our simulation is off by a factor of 25.6 % and the given simulation is off by a factor of 7.5 %.

One explanation of this difference might be the simulation time. It might take more time to form a full solvation shell with 5 water molecules. This hypothesis is supported by the fact that equilibrium was not reached in both simulations, as the energy still changes. Additionally, when testing different time ranges for the RDF calculation of the given simulation, a trend was observed that the coordination number is closer to 5, the later in the simulation we start to begin calculating the RDF. Therefore, a longer simulation would be needed to see if the

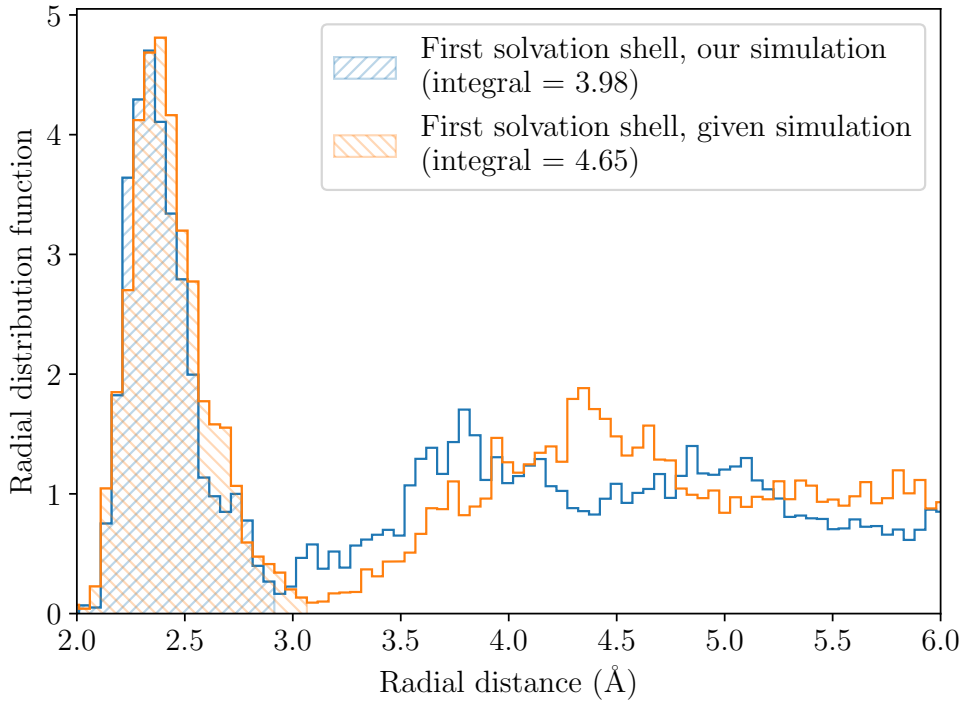


Figure 6: The two RDF's for the given simulation compared to ours. Drawn is also the area of the integral eq. (6). Observe the difference in the integral value where the given simulation has a higher value and is closer to the expected value of 5.

difference to the experimental value arises from an insufficient simulation time or if some other property of the simulation is responsible for this discrepancy.

The hydration number of 5 tells us that, for each sodium ion, a shell of 5 water molecules is needed to saturate the positive charge. This number depends on both the charge and the size of the solvent.

Task 3: Reflection and contemplation

Task 3.1: Based on this figure, which functional do you reckon is best-suited for simulating water?

There is no clear best functional shown in fig. 8, since the two experiments show different results, and it is unclear which experiment should be given more emphasis. That said, the functional "optB88-DRSLL" is in good agreement with both experimental curves, especially good with experiment 1 at the first peak in terms of height and position. The later stages are then a mixture between the two experiments eventhoug a bit of overshoot can be observed between 3.5-4.5 Å. Another good choice would be "BLYP-D3", since it is similar to a combination of both experimental curves. Where the position of the first peak agrees well with experiment 2 and the height agrees well with experiment 1. After the first peak, it is in good agreement

with both experimental curves. therefore the choice become a question of the area of interest. for peak values, optB88-DRSLL might be more suited depending on what peka you choose but for bigger distances, BLYP-D3 seems better.

Task 3.2: How well does the RDF of the given simulation agree with the experimental curves? Does it exhibit quantitative or qualitative agreement/discrepancy?

The RDF shown in fig. 7 is calculated similarly as explained in task 2.1 for the given H_2O simulation. However, this time, the distances from each oxygen atom to each oxygen atom was computed. This was done using the function "ase.atoms.get_distances", which also applies the minimum image convention so that the simulation cell is centered around the atom that is considered at each step. Used are only the last 2 ps of the simulation, since fig. 1.b) suggests that equilibrium has been reached from this point onwards.

When comparing fig. 7 to fig. 8 a qualitative, good agreement with all curves is visible, since the two peaks are positioned very similar to the experimental curves and show a similar shape. However, on a quantitative comparison, the RDF calculated here agrees very well with the blue "PBE-D3" curve shown in fig. 8.a). The extreme points reach almost the same values. This suggests that the given simulation was performed using PBE, which is the same functional we were told to use in task 1. Compared to the experimental curves, the peaks and minima are exaggerated, and the positions are shifted slightly towards smaller radii.

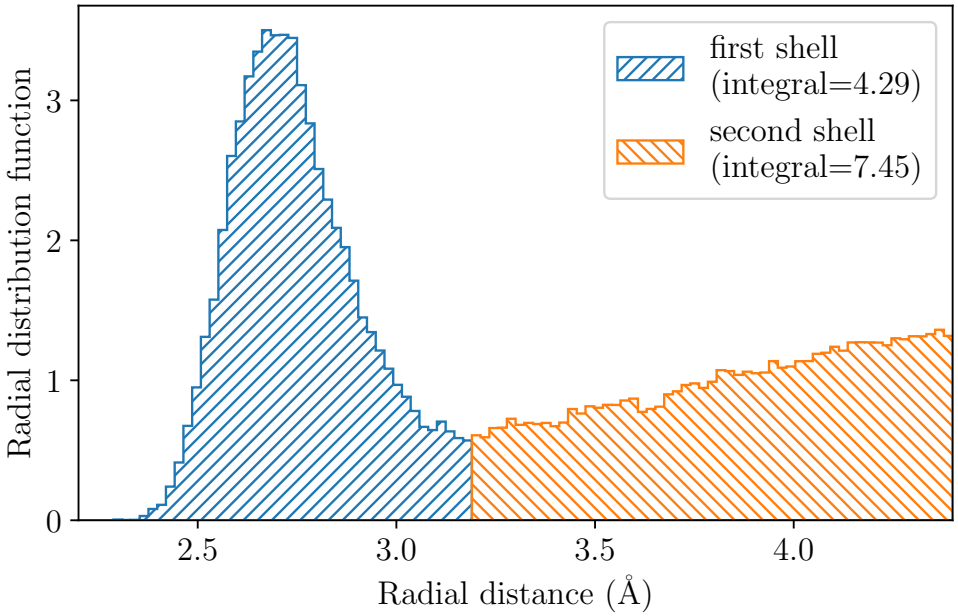
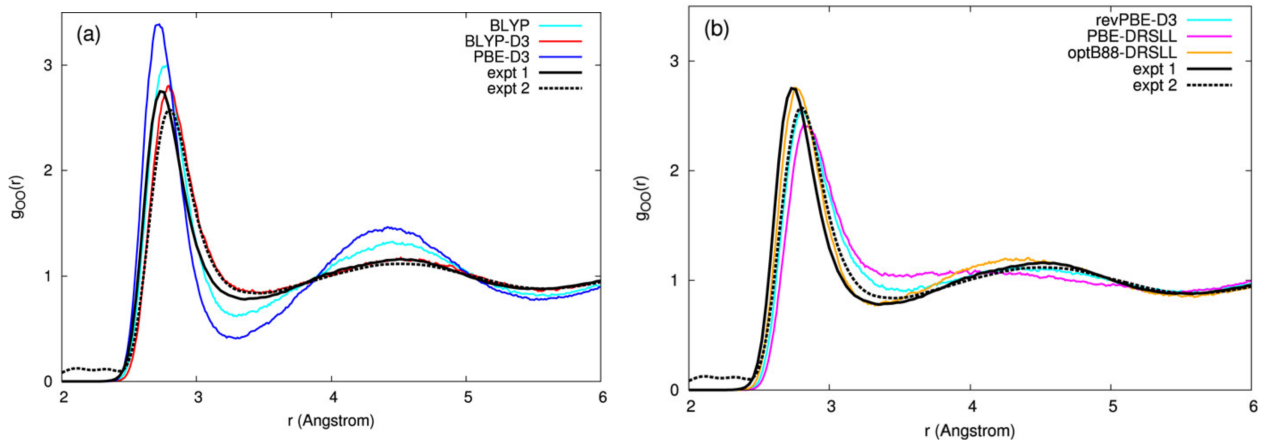


Figure 7: Radial distribution function of the last 2 ps of the given H_2O simulation. Drawn are also the integrals of the first and second shell.



(a) **(b)**

Figure 8: The RDF between oxygen atoms in water. From `https://doi.org/10.1063/1.4944633`

Task 3.3: Would a simulation cell with 7 instead of 24 molecules capture the same physics?

There are several problems with decreasing the atoms to 7 compared to 24 depending on what metric you look at. Firstly, by just looking at the coordination numbers in fig. 7, the RDF for water has a first shell of around 4 molecules and 7 molecules for the first half of the second shell. That means, in order to capture physics that arises from the dynamics of the second shell, a total of at least 18 molecules would be needed. An argument could be made that periodic boundary conditions can account for higher order shells than the first one. However, there are still losses in the symmetry of the system, which could make the behavior unphysical. If the system has important geometries inside one unit cell that need more than 7 molecules, the simulation would not be able to capture this. In other words, it is not possible to capture the same physics with only 7 molecules as with 24 molecules. However, depending on what properties and what time and length scales we are interested in 7 molecules might be enough to capture the qualitative behaviors. The size of the unit cell does also matter as the equations are using the so called bulk modulus which is determined by the size. Therefore the ratio between the unit cell and the ammount of molecules matter for the sake of density which itself is a factor for what geometeries that is possible to build and 24 molecules allows for larger cells with the same density and therefore other systems where forces might be more accuratly described.

Task 3.3: Considering an infinitely large system, why doesn't the simulation agree with experiments? What approximations are done in using this type of DFT?

Even with an infinite amount of simulated molecules, there are approximations made in AIMD that might lead to an unphysical behavior of the system. The most obvious approximation is the Born-Oppenheimer approximation, where we treat the electrons quantum mechanically

but the nuclei classically. For a light atom such as hydrogen, this approximation might not be accurate, since effects like the tunnel effect are not simulated but might contribute to some properties of water.

In the specific simulations performed here, the exchange-correlation functional PBE was used. Like every other exchange-correlation functional, this is an approximation. PBE is a non-emperical functional, which implicitly assumes that all physical constraints are already implemented in the DFT, which might lead to errors. By using PBE, it is not possible to simulate van-der-Waals interactions without introducing correction terms. This missing attractive force might lead to a density that is too low and will probably affect other properties as well. Using more accurate exchange correlation functionals like hybrid functionals might lead to more physical results. However, this would increase the computational cost by a lot.

Another approximation made in our simulation comes from using the "lcao" method. Here, the single-electron wave functions are approximated as the linear combination of atomic orbitals. This would in principle be exact, if infinitely many orbitals could be used. That said, the limitation to a finite number of orbitals can introduce errors which might even affect the physicality of the dynamics. An increased number of orbitals could make this simulation more accurate, but would also introduce more computation time.

Appendix: Python script for task 1

```
from ase.io import read
from gpaw import GPAW
from ase.md.npt import NPT
from ase.io import Trajectory

atoms = read('na_inserted.xyz')

calc = GPAW(mode='lcao',
            xc='PBE',
            basis='dzp',
            symmetry={'point_group': False}, # Turn off point-group symmetry
            charge=1, # Charged system
            txt='output.gpaw-out' # Redirects calculator output to this file
            )

atoms.set_calculator(calc)

from ase.units import fs, kB
dyn = NPT(atoms,
        temperature_K=350,
        timestep=0.5*fs, # Water scale timestep
        ttime=20*fs, # Dont forget the fs!
        pfactor=None,
        externalstress=0, # We dont use the barostat, but...
        logfile='mdOutput.log' # Outputs temperature (and more) to file at each
    timestep
        )

trajectory = Trajectory('someDynamics.traj', 'w', atoms)
```

```
dyn.attach(trajectory.write, interval=1) # Write the current positions etc. to file each timestep
dyn.run(4000) # Run 10 steps of MD simulation
```

Appendix: Python script for task 2

```
# %%
import numpy as np
import matplotlib.pyplot as plt
from tqdm.auto import tqdm

# For latex interpretation of the figures
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "Computer Modern",
    "font.size": 14.0
})

from ase.io import read, Trajectory

# %%
# Calculate the RDF
from ase.geometry import wrap_positions
nbins = 200
rmin = 0.
rmax = 10.

bin_edges = np.linspace(rmin,rmax,nbins)
bin_width = np.diff(bin_edges)[0]
bin_centers = bin_edges[:-1]+bin_width/2

def calc_rdf_of_timestep(atoms):
    elements = atoms.get_atomic_numbers()
    idx_Na = np.where(elements==11)[0]
    idxs_O = np.where(elements==8)

    pos = atoms.get_positions(wrap=True)
    pos_Na = pos[idx_Na][0]
    pos_O = pos[idxs_O]

    # implement periodic boundary conditions by copying the O positions 26 times around the main cell
    cell = atoms.get_cell()
    pbc = atoms.get_pbc()
    pos_all_O = [wrap_positions(pos_O,cell,pbc,center=[0.5+ix,0.5+iy,0.5+iz])
                 for ix in range(-1,2)
                 for iy in range(-1,2)
                 for iz in range(-1,2)]

    pos_all_O = np.concatenate(pos_all_O, axis=0)

    dists = (pos_Na[0]-pos_all_O[:,0])**2
    dists += (pos_Na[1]-pos_all_O[:,1])**2
    dists += (pos_Na[2]-pos_all_O[:,2])**2
    dists = np.sqrt(dists)
```

```
49
50      bin_counts = np.histogram(dists, bin_edges)[0]
51
52      # calculate the radial distribution function
53      rho = bin_counts.sum()/(rmax**3*np.pi*4/3)
54      rho_local = bin_counts/(4*np.pi*bin_width*bin_centers**2)
55      g = rho_local/rho
56
57      return g, rho
58
59  # Calculate the first solvation shell
60  from scipy.signal import argrelextrema
61
62  def find_first_solvation_shell(g,bin_centers,rho):
63      bin_width = bin_centers[1]-bin_centers[0]
64      first_minimum_idx = argrelextrema(g, np.less, order=4)[0][0]
65      first_minimum = bin_centers[first_minimum_idx]
66      corr_number = np.sum(g[:first_minimum_idx])*bin_width
67
68
69      corr_number = 4*np.pi*rho*np.trapz(bin_centers[:first_minimum_idx]**2*g[:
        first_minimum_idx], bin_centers[:first_minimum_idx])
70      return first_minimum_idx, corr_number
71
72  import multiprocessing as mp
73  def calc_rdf(filepath, eq_idx=0):
74      # eq_idx determines at which index to start (after equilibration)
75      trajectory = Trajectory(filepath)[eq_idx:]
76
77      # parallelize the iteration through the trajectory
78      n_cpus = mp.cpu_count()
79      with mp.Pool(processes = n_cpus) as p:
80          results = list(tqdm(p.imap(calc_rdf_of_timestep, trajectory), total=len
        (trajectory)))
81
82      g_arr = []
83      rho_arr = []
84      for res in results:
85          g_arr.append(res[0])
86          rho_arr.append(res[1])
87
88      g = np.mean(g_arr, axis=0)
89      rho = np.mean(rho_arr)
90
91      print(rho)
92
93      first_minimum_idx, corr_number = find_first_solvation_shell(g, bin_centers,
        rho)
94
95      return g, first_minimum_idx, corr_number
96
97  # %%
98  g_our, first_minimum_idx_our, corr_number_our = calc_rdf("../task1/someDynamics
        .traj")
99  g_given, first_minimum_idx_given, corr_number_given = calc_rdf("../NaCluster24.
        traj", 6000)
```

```

# %%
# Plot the RDF
plt.figure()

plt.hist(bin_centers[:first_minimum_idx_our],bin_edges[:first_minimum_idx_our
    +1],weights=g_our[:first_minimum_idx_our], histtype="step", edgecolor="C0",
    alpha=0.3, hatch="////", label=f"First solvation shell, our simulation \n(
    integral = {corr_number_our:.2f})")
plt.hist(bin_centers,bin_edges,weights=g_our, histtype="step", color="C0")

plt.hist(bin_centers[:first_minimum_idx_given],bin_edges[:
    first_minimum_idx_given+1],weights=g_given[:first_minimum_idx_given],
    histtype="step", edgecolor="C1", alpha=0.3, hatch=r"\\\\", label=f"First
    solvation shell, given simulation \n(integral = {corr_number_given:.2f})")
plt.hist(bin_centers,bin_edges,weights=g_given, histtype="step", color="C1")

plt.xlim(2,6)
plt.ylim(bottom=0)
plt.xlabel("Radial distance (Å)")
plt.ylabel("Radial distribution function")
plt.legend()
plt.tight_layout()
plt.savefig("../plots/Plottask2a3.pdf")

# %%
```

Appendix: Python script for task 3

```
# %%
import numpy as np
import matplotlib.pyplot as plt
from tqdm.auto import tqdm

from ase.io import Trajectory

# For latex interpretation of the figures
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "Computer Modern",
    "font.size": 14.0
})

trajectory = Trajectory("../cluster24.traj")[-6000:]

# %%

nbins = 200
rmin = 0.
rmax = 4.4

bin_edges = np.linspace(rmin,rmax,nbins+1)
bin_width = bin_edges[1]-bin_edges[0]
bin_centers = bin_edges[:-1]+bin_width/2

```

```
def calc_rdf_of_timestep(atoms):
    bin_counts = np.zeros_like(bin_centers)

    elements = atoms.get_atomic_numbers()
    idxs_O = np.where(elements==8)[0]

    for i in idxs_O:
        idxs_others = np.concatenate([idxs_O[:i],idxs_O[i+1:]])
        dist = atoms.get_distances(i, idxs_others, mic=True)

        bin_counts += np.histogram(dist, bin_edges)[0]

    # calculate the radial distribution function
    rho = bin_counts.sum()/(rmax**3*np.pi*4/3)
    rho_local = bin_counts/(4*np.pi*bin_width*bin_centers**2)
    rdf = rho_local/rho

    return rdf

import multiprocessing as mp
n_cpus = mp.cpu_count()
with mp.Pool(processes = n_cpus) as p:
    results = list(tqdm(p.imap(calc_rdf_of_timestep, trajectory), total=len(
    trajectory)))

print(np.array(results).shape)
rdf = np.mean(results, axis=0)

# calculate the correlation number
cut = 3.2

rho = 24/trajectory[-1].get_cell().volume
mask = bin_centers<cut
bin_centers_ = bin_centers[mask]
rdf_ = rdf[mask]
corr_number_first = 4*np.pi*rho*np.trapz(bin_centers_**2*rdf_, bin_centers_)

bin_centers__ = bin_centers[~mask]
rdf__ = rdf[~mask]
corr_number_second = 4*np.pi*rho*np.trapz(bin_centers__**2*rdf__, bin_centers__
    )

# %%
plt.figure()
plt.hist(bin_centers[mask], bin_edges, weights=rdf[mask], histtype="step",
    hatch="////", edgecolor="C0", label=f"first shell\n(integral={
    corr_number_first:.2f})")
plt.hist(bin_centers[~mask], bin_edges, weights=rdf[~mask], color="C1",
    histtype="step", hatch=r"\\\\", edgecolor="C1", label=f"second shell\n(
    integral={corr_number_second:.2f})")
plt.xlabel("Radial distance (Å)")
plt.ylabel("Radial distribution function")
plt.xlim(2.2,4.4)
plt.legend()
plt.tight_layout()
plt.savefig("../plots/h2o_rdf.pdf")
```

```
77 # %%
```