Simulating an Interesting Solution

Introduction to Molecular Modelling 2MMN40

Final Project

Fabian Maurer 1519662
Manuel Schlenkrich 1521047

February 1, 2021

Contents

1 Introduction

In this report we simulate a water-ethanol mixture ($13.5\frac{\%v}{v}$, corresponds to $x_{EtOH} = 0.11$) using intramolecular bond forces, angle forces, torsion forces and intermolecular Lennard-Jones forces.

Water-ethanol mixtures have been found to exhibit interesting behavior depending on concentration and temperature. Through experiments and simulations several points of concentration have been identified where the mixture undergoes significant structural changes. Ethanol aggregates, termed "polymers" of ethanol have been found by Nishi et al. [3] to appear at $x_{EtOH} > 0.04$. At $x_{EtOH} > 0.08$ the presence of these "polymers" appeared to be saturated and beyond $x_{EtOH} = 0.42$ they decreased in quantity. Neat ethanol does not show any aggregation.

Juurinen et al. [1] investigated bond lengths using x-ray Compton scattering and found that at low concentrations ($x_{EtOH} < 0.05$) all the O-H bonds are elongated while at high ethanol concentrations ($x_{EtOH} = 0.15 - 0.73$) they contract, increasing overall density. This suggests that a structural change of the mixture appears around $x_{EtOH} = 0.05 - 0.15$.

According to findings by Mijakovic et al. [2], aggregation of ethanol molecules happens at $x_{EtOH} < 0.2$ and weak water clustering appears at $x_{EtOH} > 0.8$ with a bicontinuous phase around $x_{EtOH} = 0.5$.

The freezing behavior of this mixture has been examined by Takaizumi [4]. Up to $x = 0.17$ the water forms ice first while beyond that point ethanol solidifies first by forming hydrates.

Our simulation only takes a subset of intermolecular interactions into account. This serverely limits the diversity of phenomena that can be observed. We can not observe any interaction between molecules relying on types of intermolecular interaction other than Lennard-Jones forces (e.g. hydrogen bonds), which means that formation of solid ice, ethanol hydrates or ethanol "polymers" will not be observable. The only phenomenon that can properly be observed is diffusion of molecules in a fluid-like manner and possibly clustering of ethanol molecules due to their higher overall weight.

2 Theory

2.1 Formulas

In this section we list the gradient formulas we used for our implementation. Formulas that were given directly in the project instructions are not listed here.

2.1.1 Bond forces

2 atoms a,b form a bond.

$$\overrightarrow{f_a} = -2k(r - r_0) * \overrightarrow{u} \tag{1}$$

$$\overrightarrow{f_b} = -\overrightarrow{f_a} \tag{2}$$

2.1.2 Angle forces

3 atoms a,b,c form an angle with b being in the middle.

$$\overrightarrow{p_a} = norm(\overrightarrow{ba} \times (\overrightarrow{ba} \times \overrightarrow{bc})) \tag{3}$$

$$\overrightarrow{f_a} = -2k\frac{\Theta - \Theta_0}{|ab|}\overrightarrow{p_a} \tag{4}$$

$$\overrightarrow{p_c} = norm(\overrightarrow{cb} \times (\overrightarrow{ba} \times \overrightarrow{bc})) \tag{5}$$

$$\overrightarrow{f_c} = -2k\frac{\Theta - \Theta_0}{|bc|}\overrightarrow{p_c} \tag{6}$$

$$\overrightarrow{f_b} = -\overrightarrow{f_a} - \overrightarrow{f_c} \tag{7}$$

2.1.3 Dihedral forces

4 atoms a,b,c,d form a dihedral with a,d being on the outer edges and b,c being in the middle.

$$\overrightarrow{p_1} = norm(\overrightarrow{ba} \times \overrightarrow{bc}) \tag{8}$$

$$\overrightarrow{f_a} = \frac{0.5}{|ab|\, sin(\Theta_1)} \left(A_1 sin(\Theta) - 2A_2 sin(2\Theta) + 3A_3 sin(3\Theta) - 4A_4 sin(4\Theta)\right) \overrightarrow{p_1} \tag{9}$$

$$\overrightarrow{p_2} = norm(\overrightarrow{cd} \times \overrightarrow{cb}) \tag{10}$$

$$\overrightarrow{f_d} = \frac{0.5}{|cd|\, sin(\Theta_1)} \left(A_1 sin(Theta) - 2A_2 sin(2\Theta) + 3A_3 sin(3\Theta) - 4A_4 sin(4\Theta)\right) \overrightarrow{p_2} \tag{11}$$

$$o = \frac{a+b}{2} \tag{12}$$

$$\overrightarrow{t_c} = -\overrightarrow{oc} \times \overrightarrow{f_d} + 0.5\overrightarrow{cd} \times \overrightarrow{f_d} + 0.5\overrightarrow{ba} \times \overrightarrow{f_a} \tag{13}$$

$$\overrightarrow{f_c} = \left(\frac{1}{|oc|^2} \overrightarrow{t_c} \times \overrightarrow{oc}\right) \tag{14}$$

$$\overrightarrow{f_c} = -\overrightarrow{f_a} - \overrightarrow{f_b} - \overrightarrow{f_d} \tag{15}$$

2.1.4 Lennard-Jones Forces

Lennard-Jones Forces apply between any 2 atoms a,b that are not part of the same molecule.

$$\overrightarrow{f_a} = \frac{24\epsilon}{r} \left(2\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right) \overrightarrow{u} \tag{16}$$

$$\overrightarrow{f_b} = -\overrightarrow{f_a} \tag{17}$$

2.2 Physical properties

The following two types of molecules are subject of the analysis in this report. In Figure 1 the physical structure of a water molecule is shown. It consists of one oxygen atom (O), which is represented in red, and two hydrogen atoms (H), which are represented in white. The molecule has two bonds, that connect each of the hydrogen atoms with the oxygen. There is one angle formed with O being the middle atom of the angle.

Figure 2 shows an ethanol molecule. It consists of two carbon atoms (C), one oxygen atom (O) and six hydrogen atoms (H). Carbon is shown in gray, oxygen in red and hydrogen in white.

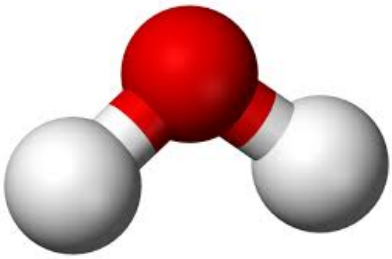

Figure 1: Structure of a water molecule

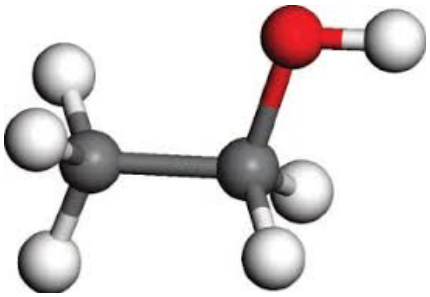

Figure 2: Structure of an ethanol molecule

2.3 Integrators

2.3.1 Euler Algorithm

The first approach to integrate the system numerically is the Euler Algorithm, which works as a forward difference in v.

Position update:

$$\underline{q_i}(t + \Delta t) = \underline{q_i}(t) + \Delta t * \underline{v}(t) + \frac{(\Delta t)^2}{2} * \frac{\underline{F_i}(t)}{m_i} \tag{18}$$

Velocity update:

$$\underline{v_i}(t + \Delta t) = \underline{v_i}(t) + \Delta t * \frac{\underline{F_i}(t)}{m_i} + \dots \tag{19}$$

2.3.2 Velocity Verlet Algorithm

A more advanced Integrator is the Velocity Verlet Algorithm.

Position update:

$$\underline{q_i}(t + \Delta t) = \underline{q_i}(t) + \Delta t * \underline{v}(t) + \frac{(\Delta t)^2}{2} * \frac{\underline{F_i}(t)}{m_i} \tag{20}$$

Velocity update:

$$\underline{v_i}(t + \Delta t) = \underline{v_i}(t) + \frac{\Delta t}{2} * \frac{\underline{\underline{F_i}}(t) + \underline{\underline{F_i}}(t + \Delta t)}{m_i} \tag{21}$$

2.3.3 Comparison

The Euler integration method always tends to 'overshoot' since it always takes a tangential path, which is merely a rough approximation as long as there is any curvature in the trajectories. This typically results in a closed system being skewed towards an increase in total energy.

The Velocity Verlet method combats this issue by computing the average of the current step and the next one. This eliminates a large part of the error. While there is still some error, energy preservation is maintained to a large degree in a closed system.

3 Simulation

3.1 Single Water Molecule

In order to check the MDS code for correctness a system containing only one water molecule is analysed as a first step. Molecule properties, such as bond lengths and angle are successively tested to get an impression, if the resulting movements are reasonable.

Figure 3 shows the oscillation of the two bond lengths around the equilibrium length $r_0 = 0.0957$nm.

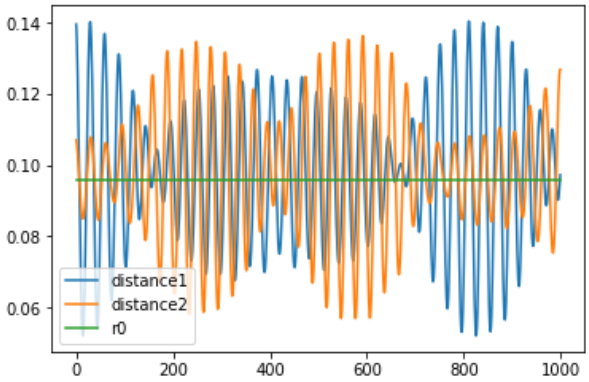

Figure 3: Bond lengths in a water molecule over time

Figure 4 shows the behaviour of the angle, if only angle forces are taken into account. One clearly sees that the angle oscillates around the equilibrium angle θ_0 of 104.52° (= 1.82). However as time passes the angle seems to converge to the equilibrium angle, which is a result of the distance between the oxygen and hydrogen atoms getting bigger over time. This phaenomen could not really be explained by us. However if additional to the angle forces also bond forces are taken into account the angle behaves as shown in Figure 5. The bond force keeps the distance between the oxygen and hydrogen atoms bounded and therefore the angle oscillates regularly around the equilibrium angle θ_0.

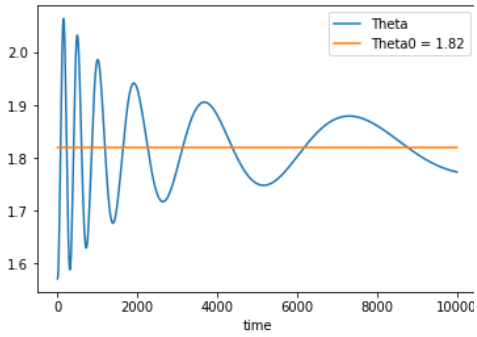

Figure 4: Angle of Water Molecule only affected by angle forces

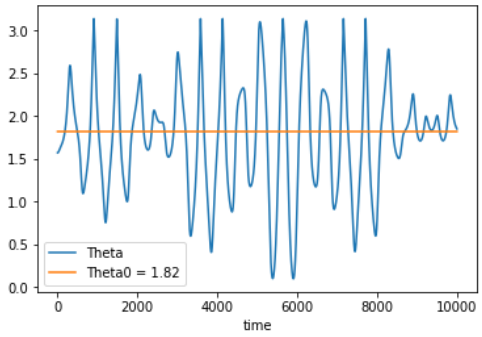

Figure 5: Angle of Water Molecule affected by angle- and bond forces

3.2 A system of pure water

3.2.1 Setup

We simulated a total of 903 H_2O molecules in a box of size 3nm x 3nm x 3nm. Molecules were distributed on a cartesian grid with some amount of random permutation (25% of the mesh size). Temperature was set to a range of [250k,350k]. Periodic boundaries were used. Contrary to the environment outlined in the project description, we chose to use a timestep interval that made the most sense. If the interval is too short, only bond forces can be properly observed while Lennard-Jones-Forces barely have an effect. If the interval is too long, the system becomes unstable since bond forces become too large for the integrator to handle. So we empirically chose the largest possible timestep interval where the system was still stable.

3.2.2 Result

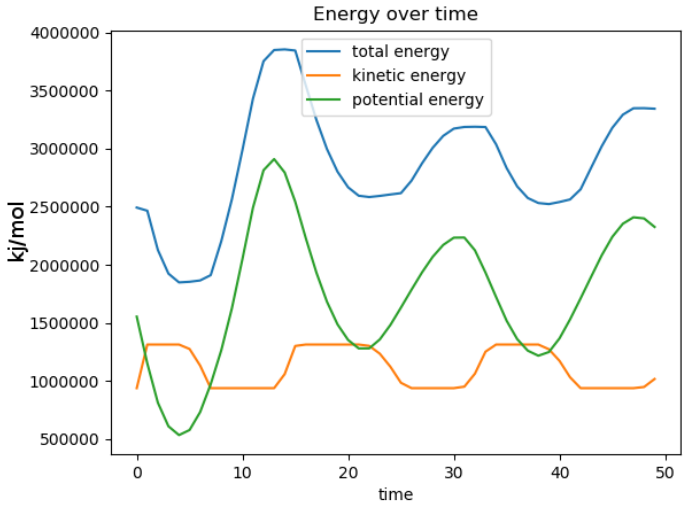

Figure 6: Energy of 903 Water Molecules at [250k,350K]

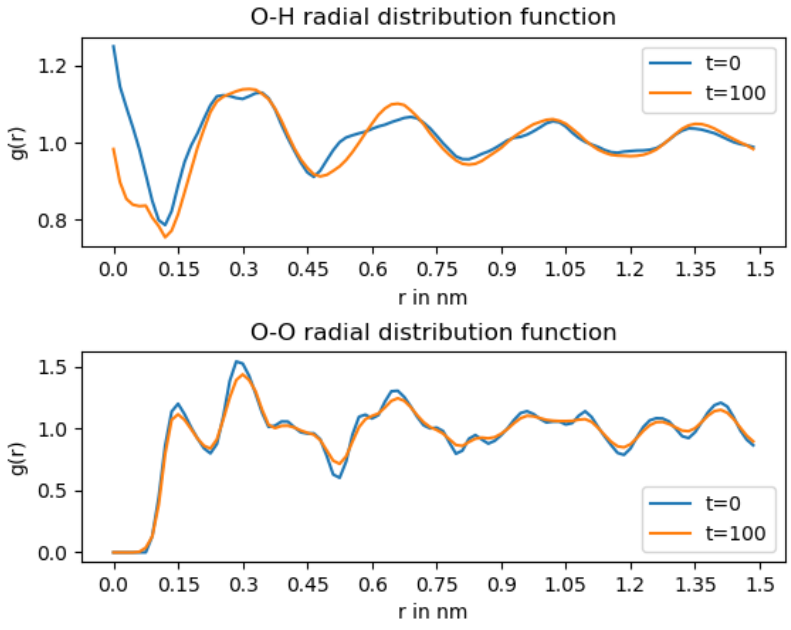

Figure 7: Radial Distribution of O-H and O-O in pure water at [250K,350K]

3.2.3 Discussion

In this system of pure water we observed reconfiguration towards a more energetically efficient molecule mesh. Our initial condition of placing the molecules on a cartesian grid is not optimal, which shows in the radial distribution function that becomes a lot smoother after 100 timesteps. The kinetic energy is constrained to a certain range since we use a temperature range which which creates those sudden bends in both the kinetic and total energy of the system. While the system is closed and would therefore normally not experience a change in total energy, our way of setting the temperature does not allow for that. In sections where the temperature is allowed to move freely between 250K and 350K without adjustment, the total energy stays pretty much the same. That is what we expected to happen.

3.3 A system of pure ethanol

3.3.1 Setup

We simulated a total of 280 C_2H_6O molecules in a box of size 3nm x 3nm x 3nm. Temperature was set to a range of [250k,350k]. Periodic boundaries were used. We chose our timestep interval in the same way we did in 3.2 .

3.3.2 Result

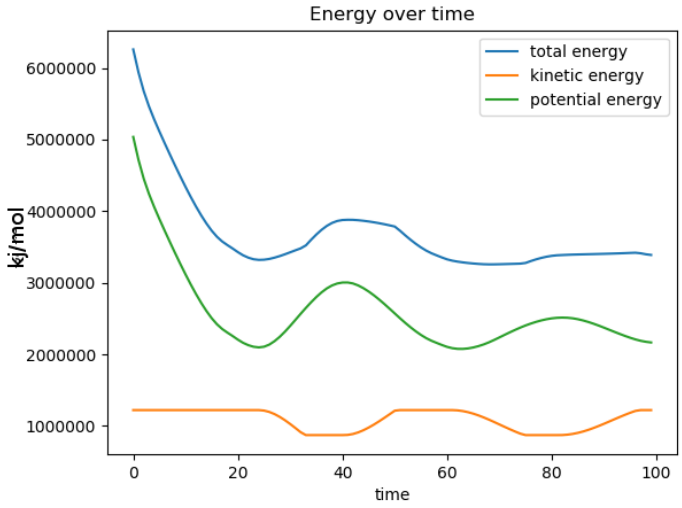

Figure 8: Energy of 280 Ethanol Molecules at 300K

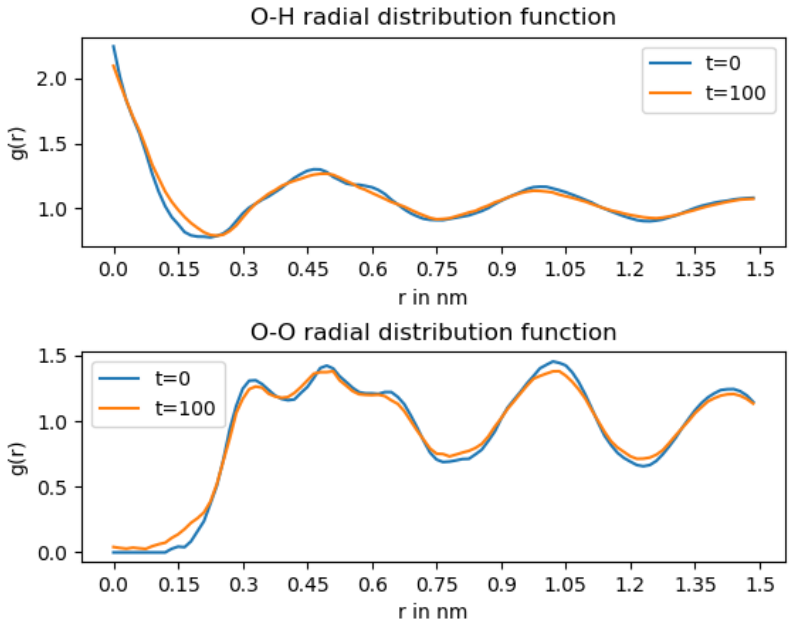

Figure 9: Radial Distribution of O-H and O-O in pure ethanol at [250K,350K]

3.3.3 Discussion

The result is very similar to a system of pure water since our simulation techniques do not take any intermolecular effects beyond lennard-jones forces into account. The molecules are larger and heavier than water molecules, which results in less molecules in the same area at the same temperature. The same kind of reconfiguration can be observed when looking at the radial distribution function.

3.4 A system of 13.5 %v/v ethanol in water

3.4.1 Setup

We simulated a total of 781 H_2O molecules and 37 C_2H_6O molecules in a box of size 3nm x 3nm x 3nm. Temperature was set to a range of [250k,350k]. Periodic boundaries were used. Once again we chose our timestep interval in the same way we did in 3.2 .

3.4.2 Result

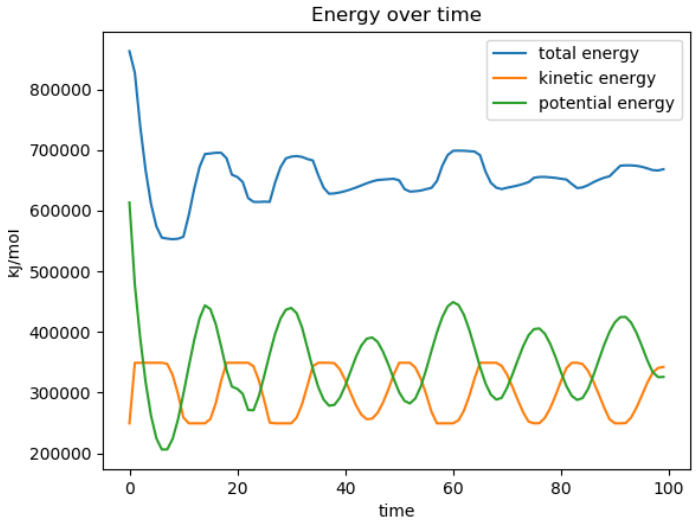

Figure 10: Energy of a system of 13.5% ethanol in water at [250K,350K]

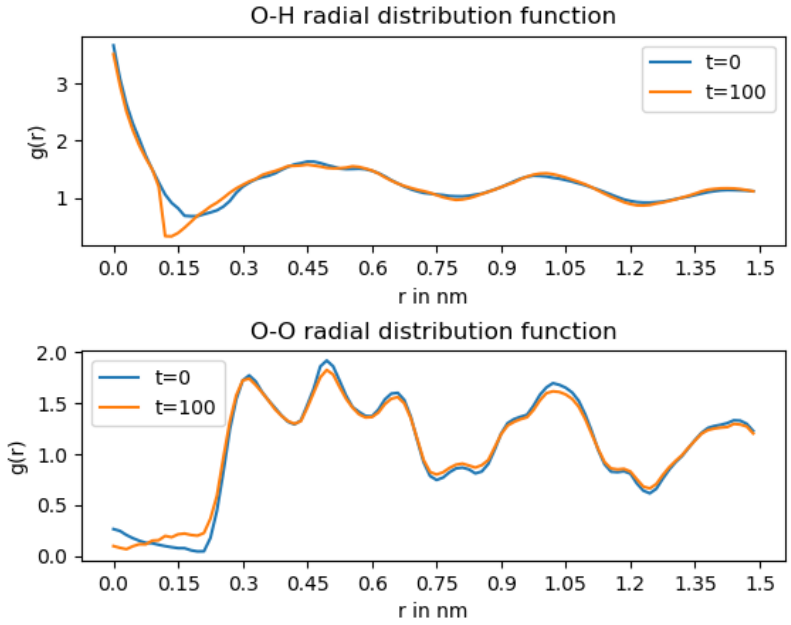

Figure 11: Radial Distribution of O-H and O-O in a system of 13.5% ethanol in water at [250K,350K]

3.4.3 Discussion

We observed diffusion effects when visualizing our results. Our initial condition of an equidistant cartesian grid proved to be least in this one out of all setups. Ethanol molecules started out quite close to neighboring molecules, but it appears that their stronger repulsion forces (compared to H2O) leads to a system where the distance of other molecules to ethanol molecules is considerably larger than the distance to water molecules. Interestingly, there is a sharp downwards spike in the O-H radial distribution function. This happened with any initial conditions we tried. We could not observe any clustering of either kind of molecule.

4 Code

The MDS Code, which is used to simulate and analyze the behaviour of the solution treated in this report is written in Python 3 and consists of the following main functionalities, shown in Figure 12.

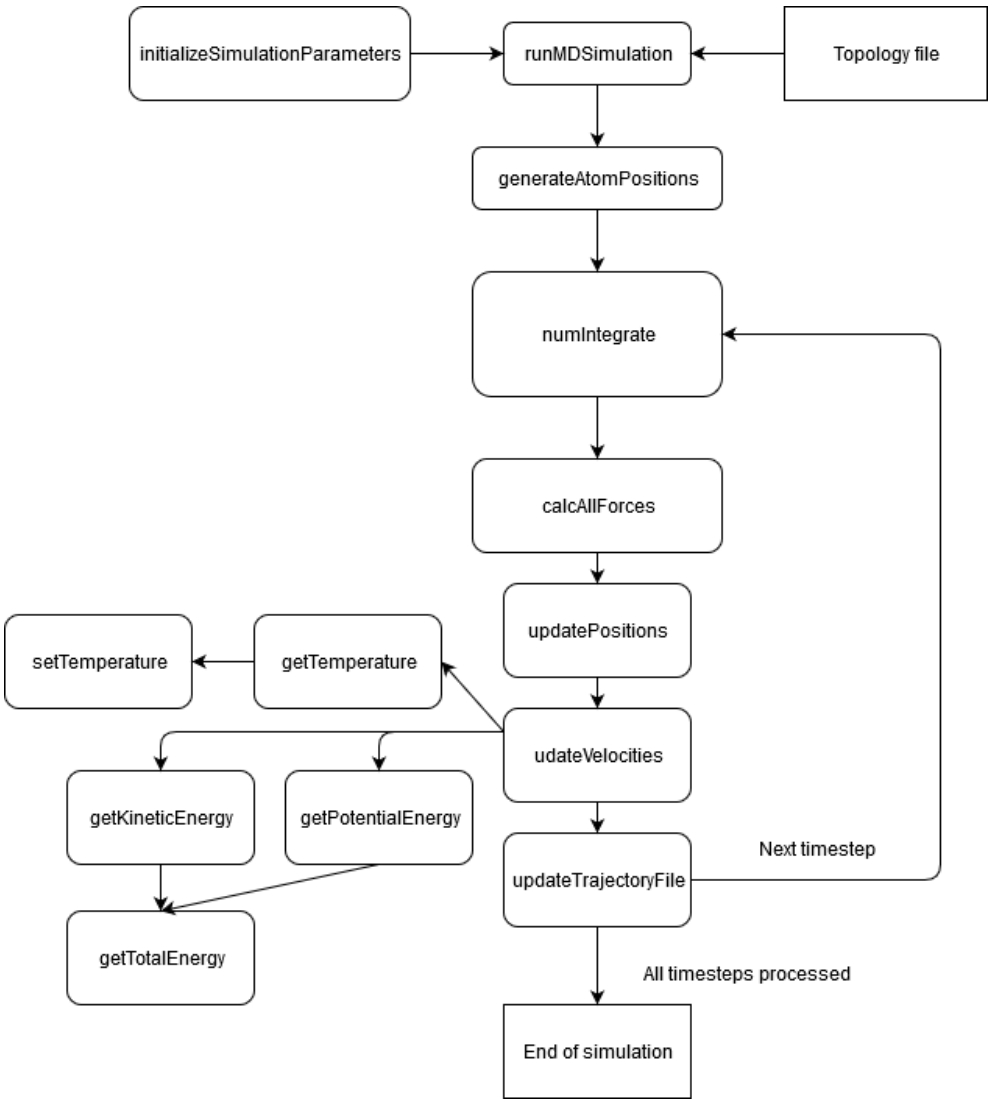

Figure 12: Flow Chart of MDS Code

The Code is organized in a way, that the whole simulation can be run by one overall function called *runMDSimulation*. This function can be adjusted by the

user via the parameters *topology_filename*, *integrator*, *timestep*, *nr_timesteps*, *cutoff*, *boxsize*, *margin*, *targettemp*. These parameters basically define the whole simulation and therefore the code is very flexible and no major changes within the code have to be made.

The code makes extensive use of the *numpy* library in order to deal with big vectors and matrices caused by a large number of atoms and timesteps.

5 Conclusion

We successfully implemented a basic molecular dynamics simulation and used it to simulate the behavior of a mixture of water and ethanol. The effects observed were mostly due to Lennard-Jones-Forces, i.e. diffusion effects. This is in line with what we expected, since we are not simulating any topological changes of the molecules such as new bonds forming. We used a radial distribution function among other measures to support the validity of our implementation.

6 Appendix

6.1 MDS Code

```
import numpy as np
from topology import Molecule
from topology import Topology
import matplotlib.pyplot as plt
import csv
import random
import time

#########################################################################
def ReadTrajectory(trajFile):
    trajectory=[]
    with open(trajFile, "r") as tF:
        line = tF.readline()
        while line is not "":
            #first line is number of atoms
            N = int(line.strip())
            tF.readline().strip()
            q = []
            for i in range(N):
                line = tF.readline().strip().split(" ")
                for c in line[1:]:
                    if c is not "":
                        q.append(float(c))
            trajectory.append(np.array(q))

            line = tF.readline()

    return trajectory, N

#########################################################################
```

```
def calcDistance(vectors,n):
    r2 = np.linalg.norm(vectors, axis=2)
    return r2

#########################################################################
def calcBondPotential_MultipleAtoms(top, xyzarray):
    indexr1=top.getBond_i_indeces()
    indexr2=top.getBond_j_indeces()

    coord_r1=xyzarray[indexr1]
    coord_r2=xyzarray[indexr2]

    r0=top.getBond_r0()
    k=top.getBond_k()

    return(calcBondPotential(coord_r1,coord_r2,r0,k))

#########################################################################
def calcBondPotential(r1, r2, r0, k):
    r=np.linalg.norm(r1-r2, axis=1)
    potential_atom_atom=0.5*k*(r-r0)**2
    return potential_atom_atom

#########################################################################
def calcBondForce_MultipleAtoms(top,xyzarray):
    indexr1=top.getBond_i_indeces()
    indexr2=top.getBond_j_indeces()

    coord_r1=xyzarray[indexr1]
    coord_r2=xyzarray[indexr2]

    r0=top.getBond_r0()
    k=top.getBond_k()
```

```
    return(calcBondForce(coord_r1,coord_r2,r0,k))

#######################################################################
def calcBondForce(r1, r2, r0, k):
    r=np.linalg.norm(r1-r2, axis=1)
    force1 = np.transpose((np.transpose(r1-r2)/r)*(-2*k*(r-r0)))
    force2 = -force1

    return np.reshape(np.array([force1, force2]),(2*len(r0),3))
    #ATTENTION
    #return array is shaped like
    #[force on r1 of first bond]
    #[force on r1 of second bond]
    #...
    #[force on r1 of last bond]
    #[force on r2 of first bond]
    #[force on r2 of second bond]
    #...
    #[force on r2 of last bond]


#######################################################################
def calcAnglePotential_MultipleAtoms(top,xyzarray):
    indexr1=top.getAngle_i_indeces()
    indexr2=top.getAngle_j_indeces()
    indexr3=top.getAngle_k_indeces()

    coord_r1=xyzarray[indexr1]
    coord_r2=xyzarray[indexr2]
    coord_r3=xyzarray[indexr3]

    k0=top.getAngle_k0()
```

```
    theta0=top.getAngle_theta0()

    return(calcAnglePotential(coord_r1, coord_r2, coord_r3, theta0, k0 ))

#########################################################################
def calcAnglePotential(r1, r2, r3, theta0, k_theta):
    v1 = r1-r2
    v2 = r1-r3
    v1_norm = np.linalg.norm(r1-r2, axis=1)
    v2_norm = np.linalg.norm(r1-r3, axis=1)
    dots = (v1 * v2).sum(axis=1)
    theta = np.arccos(np.clip(dots/(v1_norm*v2_norm),-1,1))
    angle_potential = 0.5*k_theta*(theta-theta0)**2

    return angle_potential

#########################################################################
def calcAngleForce_MultipleAtoms(top,xyzarray):
    indexr1=top.getAngle_i_indeces()
    indexr2=top.getAngle_j_indeces()
    indexr3=top.getAngle_k_indeces()

    coord_r1=xyzarray[indexr1]
    coord_r2=xyzarray[indexr2]
    coord_r3=xyzarray[indexr3]

    k0=top.getAngle_k0()
    theta0=top.getAngle_theta0()

    return(calcAngleForce(coord_r1, coord_r2, coord_r3, theta0, k0 ))

#########################################################################
def calcAngleForce(r1, r2, r3, theta0, k_theta):
```

```
    v1 = r1-r2
    v2 = r3-r2
    v1_norm = np.linalg.norm(v1, axis=1)
    v2_norm = np.linalg.norm(v2, axis=1)
    dots = (v1 * v2).sum(axis=1)

    theta = np.arccos(np.clip(dots/(v1_norm*v2_norm),-1,1))

    p_a = np.cross(v1,np.cross(v1,v2))
    p_a = p_a/np.linalg.norm(p_a)
    f_a = np.transpose(np.transpose(p_a)*(-2*k_theta*(theta-theta0)))
    /np.linalg.norm(v1)

    p_c = np.cross(-v2,np.cross(v1,v2))
    p_c = p_c/np.linalg.norm(p_c)
    f_c = np.transpose(np.transpose(p_c)*(-2*k_theta*(theta-theta0)))
    /np.linalg.norm(v2)

    f_b = -f_a - f_c

    return np.reshape(np.array([f_a, f_b, f_c]),(3*len(theta),3))

    #pay ATTENTION
    #return array is shaped like
    #[f_a of angle 1]
    #[f_a of angle 2]
    #[f_b of angle 1]
    #[f_b of angle 2]
    #[f_c of anlge 1]
    #[f_c of angle 2]

#####################################################################
def calcAllForces(top, xyzarray, bbox, cutoff):
```

```
F = np.zeros([len(top.getElements()),3])
####
#calculate all the forces
####
bond_forces = calcBondForce_MultipleAtoms(top, xyzarray)

angle_forces = calcAngleForce_MultipleAtoms(top, xyzarray)



#if top.hasDiheadrals():
    #diheadral_forces = calcDiheadralForce_MultipleAtoms(top, xyzarray)
lennardjonesforces = calcLennardJonesForces(top, xyzarray, bbox, cutoff)
lennardjonesforces
=np.multiply(lennardjonesforces,top.getLennardJonesMatrix())
####

####
#sum all the forces up to one matrix
####

#bondforces
bond_forces_split = np.split(bond_forces, 2)

indexr1=top.getBond_i_indeces()
indexr2=top.getBond_j_indeces()

np.add.at(F,indexr1,bond_forces_split[0])
np.add.at(F,indexr2,bond_forces_split[1])
#angleforces
angle_forces_split = np.split(angle_forces, 3)

indexr1=top.getAngle_i_indeces()
indexr2=top.getAngle_j_indeces()
```

```
    indexr3=top.getAngle_k_indeces()

    np.add.at(F,indexr1,angle_forces_split[0])
    np.add.at(F,indexr2,angle_forces_split[1])
    np.add.at(F,indexr3,angle_forces_split[2])

    #diheadralforces
    #if top.hasDiheadrals():
    #    diheadral_forces_split = np.split(diheadral_forces, 4)

    #    indexr1=top.getDiheadral_i_indeces()
    #    indexr2=top.getDiheadral_j_indeces()
    #    indexr3=top.getDiheadral_k_indeces()
    #    indexr4=top.getDiheadral_l_indeces()



    #    np.add.at(F,indexr1,diheadral_forces_split[0])
    #    np.add.at(F,indexr2,diheadral_forces_split[1])
    #    np.add.at(F,indexr3,diheadral_forces_split[2])
    #    np.add.at(F,indexr4,diheadral_forces_split[3])

    #lennardjones forces

    F+=np.sum(lennardjonesforces,axis=1)

    #for i in range(0, len(top.getElements())):
        #for k in range(0,len(top.getElements())):
            #if i != k:
                # F[k] += lennardjonesforces[i][k]

    return F

##########################################################################
```

```
def numIntegrateVelocityVerlet(top, xyzarray, timestep, nr_timesteps,
                               cutoff, targetTemp):
    bbox = getbbox(xyzarray)

    velocities = np.zeros([len(top.getElements()),3])

    totalEnergyHistory=[]
    kinEnergyHistory=[]
    potEnergyHistory=[]
    for i in range(0, len(top.getElements())):
        u = np.random.uniform(size=3)
        u /= np.linalg.norm(u) # normalize
        vRand = 0.01*u
        velocities[i] = vRand
    with open("output.xyz", "w") as f:
        writer = csv.writer(f,delimiter=' ',lineterminator='\n')
        for j in range(0, nr_timesteps):
            print('timestep:', j)

            if j==0:
                F = calcAllForces(top, xyzarray, bbox, cutoff)

            xyzarray+=timestep*velocities
            + (((timestep)**2)/2) *np.transpose((np.transpose(F)
            /top.getMasses()))

            F_new = calcAllForces(top, xyzarray, bbox, cutoff)
            #print(np.sum(F_new))

            velocities+=(timestep/2)*np.transpose(np.transpose((F + F_new))
            /top.getMasses())

            #print(xyzarray)
```

```
            #write positions in xyz file
            writer.writerow([len(top.getElements())])
            writer.writerow(['t=']+[j])
            k = 0
            clippedPositions=applyPositionPBC(xyzarray,bbox)
            #clippedPositions = xyzarray
            for atom in clippedPositions:
                writer.writerow([top.getElements()[k]]+
                [10*atom[0]]+[10*atom[1]]+[10*atom[2]])
                k += 1

            #Temperature
            temp=getTemperature(top, velocities)
            velocities = setTemperature(velocities, temp, targetTemp)

            print("The temperature is: ", getTemperature(top, velocities))

            total, kin, pot = getEnergies(top,xyzarray,velocities,bbox,cutoff)
            totalEnergyHistory.append(total)
            kinEnergyHistory.append(kin)
            potEnergyHistory.append(pot)




            F=F_new

    printEnergyHistories(top,nr_timesteps,totalEnergyHistory,
                         kinEnergyHistory,potEnergyHistory)
    return 1

#########################################################################
def numIntegrateEuler(top, xyzarray, timestep, nr_timesteps,
```

```
                    cutoff, targetTemp):
bbox = getbbox(xyzarray)

velocities = np.zeros([len(top.getElements()),3])
for i in range(0, len(top.getElements())):
    u = np.random.uniform(size=3)
    u /= np.linalg.norm(u) # normalize
    vRand = 0.01*u
    velocities[i] = vRand
with open("output.xyz", "w") as f:
    writer = csv.writer(f,delimiter=' ',lineterminator='\n')
    for j in range(0, nr_timesteps):
        print('timestep:', j)
        #calculate all the forces
        F = calcAllForces(top, xyzarray, bbox, cutoff)

        #udate positions
        xyzarray += timestep*velocities +
        (((timestep)**2)/2) *np.transpose((np.transpose(F)
        /top.getMasses()))

        #update velocities
        velocities += timestep*np.transpose(np.transpose(F)
        /top.getMasses())

        #print(xyzarray)
        #write positions in xyz file
        writer.writerow([len(top.getElements())])
        writer.writerow(['t=']+[j])
        k = 0
        #clippedPositions=applyPositionPBC(xyzarray,bbox)
        clippedPositions = xyzarray
        for atom in clippedPositions:
```

```
                writer.writerow([top.getElements()[k]]+
                [atom[0]]+[atom[1]]+[atom[2]])
                k += 1

            #Temperature
            temp=getTemperature(top, velocities)
            velocities = setTemperature(velocities, temp, targetTemp)

            print("The temperature is: ", getTemperature(top, velocities))


    #return(positions, velocities)
    return 1

##########################################################################
def calcLennardJonesInteraction(top, xyzarray, bbox, cutoff):
    #calculate distances between all the atoms
    vectors = xyzarray - xyzarray[:, np.newaxis]
    vectors = applyVectorPBC(vectors,bbox)
    distances = calcDistance(vectors, len(top.getElements()))

    high_distance_flags = (distances > cutoff)
    distances += np.identity(len(top.getElements()))

    #print(distances)
    sigmas = top.getAtom_Sigma()
    epsilons = top.getAtom_Epsilon()

    sigmas_ij = calcSigmas_ij(top, sigmas)
    epsilons_ij = calcEpsilons_ij(top,epsilons)

    U = 4*epsilons_ij*((sigmas_ij/distances)**12-(sigmas_ij/distances)**6)
```

```
    U1 = 4*epsilons_ij
    np.fill_diagonal(U1,0.0)
    U2 = (sigmas_ij/distances)
    np.fill_diagonal(U2,0.0)

    U1[high_distance_flags]=0.0

    U = U1*(U2**12-U2**6)

    np.fill_diagonal(U,0.0)
    U=np.multiply(U,top.getLennardJonesMatrix2d())
    return U

##########################################################################
def calcLennardJonesForces(top, xyzarray, bbox, cutoff):
    vectors = xyzarray - xyzarray[:, np.newaxis]

    #vectors = [applyVectorPBC(x,bbox) for x in vectors]

    #vectors = np.apply_along_axis(applyPBC,0,vectors,bbox)
    vectors = applyVectorPBC(vectors,bbox)
    #calculate distances between all the atoms

    distances = calcDistance(vectors, len(top.getElements()))

    high_distance_flags = (distances > cutoff)
    #low_distance_flags = (distances < 0.2)
    distances += np.identity(len(top.getElements()))
    #distances[low_distance_flags]=0.2

    sigmas = top.getAtom_Sigma()
    epsilons = top.getAtom_Epsilon()
```

```
    sigmas_ij = calcSigmas_ij(top, sigmas)
    epsilons_ij = calcEpsilons_ij(top,epsilons)

    U1 = ((24*epsilons_ij)/distances)
    np.fill_diagonal(U1,0.0)
    U2 = (sigmas_ij/distances)
    np.fill_diagonal(U2,0.0)

    U1[high_distance_flags]=0.0

    U = U1*(2*U2**12-U2**6)

    np.fill_diagonal(U,0.0)

    vectors_normed = np.transpose(np.transpose(vectors)/distances)

    forces = -np.transpose(U*np.transpose(vectors_normed))

    return forces

##########################################################################
def calcSigmas_ij(top,sigmas):
    trajectory = np.array([np.concatenate(sigmas, axis=None)])
    for s,q in enumerate(trajectory):
        sigmas_ij = 0.5*(q + q[:, np.newaxis] )
    return sigmas_ij

##########################################################################
def calcEpsilons_ij(top,epsilons):
    trajectory = np.array([np.concatenate(epsilons, axis=None)])
    for s,q in enumerate(trajectory):
        epsilons_ij = np.sqrt((q * q[:, np.newaxis]))
    return epsilons_ij;
```

```
#########################################################################
def calcDiheadralPotential_MultipleAtoms(top,xyzarray):
    indexr1=top.getDiheadral_i_indeces()
    indexr2=top.getDiheadral_j_indeces()
    indexr3=top.getDiheadral_k_indeces()
    indexr4=top.getDiheadral_l_indeces()



    coord_r1=xyzarray[indexr1]
    coord_r2=xyzarray[indexr2]
    coord_r3=xyzarray[indexr3]
    coord_r4=xyzarray[indexr4]

    C1=top.getDiheadral_C1()
    C2=top.getDiheadral_C2()
    C3=top.getDiheadral_C3()
    C4=top.getDiheadral_C4()

    return(calcDiheadralPotential(coord_r1, coord_r2, coord_r3, coord_r4,
                                  C1, C2, C3, C4))

#########################################################################
def calcDiheadralPotential(coord_r1, coord_r2, coord_r3, coord_r4,
                                  C1, C2, C3, C4):
    v1 = coord_r2 - coord_r1
    v2 = coord_r2 - coord_r3

    v3 = coord_r3 - coord_r2
    v4 = coord_r3 - coord_r4

    normal1 = np.cross(v1,v2)
    normal2 = np.cross(v3,v4)
```

```
    normal1_norm = np.linalg.norm(normal1, axis=1)
    normal2_norm = np.linalg.norm(normal2, axis=1)
    dots = (normal1 * normal2).sum(axis=1)
    theta = np.arccos(np.clip(dots/(normal1_norm*normal2_norm),-1,1))

    theta -= np.pi

    P = 0.5*(C1*(1+np.cos(theta))+C2*(1-np.cos(2*theta))+C3*(1+np.cos(3*theta))

    return P


##########################################################################
def calcDiheadralForce_MultipleAtoms(top, xyzarray):
    indexr1=top.getDiheadral_i_indeces()
    indexr2=top.getDiheadral_j_indeces()
    indexr3=top.getDiheadral_k_indeces()
    indexr4=top.getDiheadral_l_indeces()



    coord_r1=xyzarray[indexr1]
    coord_r2=xyzarray[indexr2]
    coord_r3=xyzarray[indexr3]
    coord_r4=xyzarray[indexr4]

    C1=top.getDiheadral_C1()
    C2=top.getDiheadral_C2()
    C3=top.getDiheadral_C3()
    C4=top.getDiheadral_C4()

    return(calcDiheadralForce(coord_r1, coord_r2, coord_r3, coord_r4,
                              C1, C2, C3, C4))
```

```
#########################################################################
def calcDiheadralForce(coord_r1, coord_r2, coord_r3, coord_r4,
                       C1, C2, C3, C4):
    v1 = coord_r2 - coord_r1
    v2 = coord_r2 - coord_r3

    v3 = coord_r3 - coord_r2
    v4 = coord_r3 - coord_r4

    normal1 = np.cross(v1,v2)
    normal2 = np.cross(v3,v4)

    normal1_norm = np.linalg.norm(normal1, axis=1)
    normal2_norm = np.linalg.norm(normal2, axis=1)
    dots = (normal1 * normal2).sum(axis=1)
    theta = np.arccos(np.clip(dots/(normal1_norm*normal2_norm),-1,1))

    theta -= np.pi

    V = 0.5*(C1*(-np.sin(theta))+C2*(2*np.sin(2*theta))+
    C3*(3*np.sin(3*theta))+C4*(4*np.sin(4*theta)))

    ##################
    #angles between r1,r2,r3
    v1_norm = np.linalg.norm(v1, axis=1)
    v2_norm = np.linalg.norm(v2, axis=1)
    dots = (v1 * v2).sum(axis=1)
    theta_1 = np.arccos(np.clip(dots/(v1_norm*v2_norm),-1,1))

    ##################
    #angles between r2,r3,r4
    v3_norm = np.linalg.norm(v3, axis=1)
```

```
    v4_norm = np.linalg.norm(v4, axis=1)
    dots = (v3 * v4).sum(axis=1)
    theta_2 = np.arccos(np.clip(dots/(v3_norm*v4_norm),-1,1))

    f_1 = np.transpose((1/(v1_norm*np.sin(theta_1))*
    V*np.transpose(normal1)/normal1_norm))

    f_4 = np.transpose((1/(v4_norm*np.sin(theta_2))*
    V*np.transpose(normal2)/normal2_norm))

    c_mid = (coord_r3 + coord_r2)/2
    v_mid = coord_r3 - c_mid
    v_mid_norm = np.linalg.norm(v_mid, axis=1)

    t_c = -(np.cross(v_mid,f_4)+np.cross(0.5*v4,f_4)+np.cross(-v1,f_1))

    f_3 = np.cross(np.transpose(np.transpose(t_c)*((1/v_mid_norm)**2)), v_mid)

    f_2 = -f_1-f_3-f_4

    return np.reshape(np.array([f_1, f_2, f_3, f_4]),(4*len(theta),3))
##########################################################################
def getTemperature(top, velocities):
    #STILL TO DO
    k_B = 0.008314
    N_f = 3*len(top.getElements())

    #kinetic energy of one atom k = 0.5*m*v^2
    kinetic = 0.5*top.getMasses()*
    np.linalg.norm(velocities*(10^-9),axis=1)**2/1000
    E_kin = np.sum(kinetic)

    T = (2*E_kin)/(k_B*N_f)
```

```
    return T

##########################################################################
def setTemperature(velocities,temp,targetTemp):
    velocities*=np.sqrt(targetTemp/temp)
    return velocities

##########################################################################
def rescaleVelocity(v_old, v_new):
    v_new_normed = np.transpose(np.transpose(v_new)/np.linalg.norm(v_new, axis=

    v_rescaled = np.transpose(np.transpose(v_new_normed)*
    np.linalg.norm(v_old, axis=1))

    return v_rescaled

##########################################################################
def getbbox(points):
    a = np.max(points, axis=0)
    return a

##########################################################################
def applyVectorPBC(vectors,bbox):
    vectors=np.array([modifiedAbs(a,bbox[0]) for a in vectors])
    return vectors

##########################################################################
def modifiedAbs(n,div):
    return np.where(n%div>(div/2),n%div-div,n%div)

##########################################################################
def applyPositionPBC(points,bbox):
    a = np.zeros(points.shape)
```

```
    a[:,0]=[(x%bbox[0]) for x in points[:,0]]
    a[:,1]=[(x%bbox[1]) for x in points[:,1]]
    a[:,2]=[(x%bbox[2]) for x in points[:,2]]
    return a

##########################################################################
def generateUniformAtoms(box,margin,top):
    atoms=[]
    molecules=top.getMolecules()

    c=0
    randomiser=np.arange(len(molecules))
    np.random.shuffle(randomiser)
    perRow=np.ceil(np.cbrt(len(molecules)))
    baseDist=box[0]/perRow

    for m in molecules:
        count=randomiser[c]
        x=(count%perRow)*baseDist+baseDist/2+random.random()*baseDist/4
        y=(np.floor(count/perRow)%perRow)*baseDist+baseDist/2
        z=(np.floor(count/(perRow**2)))*baseDist+baseDist/2
        if m.name=='Water':
            atoms.append([x,y,z])
            atoms.append([x+0.05+random.random()*0.1,y,z+random.random()*0.1])
            atoms.append([x,y+0.05+random.random()*0.1,z+random.random()*0.1])
        elif m.name=='H2':
            atoms.append([x,y,z])
            atoms.append([x+0.05+random.random()*0.1,y+0.05+random.random()*0.1
        elif m.name=='Methane':
            atoms.append([x+0.1,y+0.1,z+0.1])
            atoms.append([x+0.1,y,z])
            atoms.append([x,y+0.1,z])
            atoms.append([x,y,z+0.1])
```

```
            atoms.append([x+0.1,y+0.1,z])
        elif m.name=='Ethanol':
            atoms.append([x+0.05+random.random()*0.1,y+0.05+random.random()*0.1
            atoms.append([x+0.15+random.random()*0.1,y+0.05+random.random()*0.1
            atoms.append([x+0.25+random.random()*0.1,y+0.05+random.random()*0.1
            atoms.append([x+0.05+random.random()*0.1,y+0.0,z+random.random()*0.
            atoms.append([x+0.15+random.random()*0.1,y+0.0,z+random.random()*0.
            atoms.append([x+0.25+random.random()*0.1,y+0.0,z+random.random()*0.
            atoms.append([x+0.05+random.random()*0.1,y+0.2,z+random.random()*0.
            atoms.append([x+0.25+random.random()*0.1,y+0.2,z+random.random()*0.
            atoms.append([x+0.25+random.random()*0.1,y+0.2,z+random.random()*0.
        else:
            print('Molecule not found ERROR')
        c+=1

    return np.array(atoms, dtype=float)

##########################################################################
def generateRandomAtoms(box,margin,top):
    atoms=[]
    molecules=top.getMolecules()

    for m in molecules:
        x=margin+random.random()*(box[0]-2*margin)
        y=margin+random.random()*(box[1]-2*margin)
        z=margin+random.random()*(box[2]-2*margin)
        if m.name=='Water':
            atoms.append([x,y,z])
            atoms.append([x+0.05+random.random()*0.1,y,z+random.random()*0.1])
            atoms.append([x,y+0.05+random.random()*0.1,z+random.random()*0.1])
        elif m.name=='H2':
            atoms.append([x,y,z])
            atoms.append([x+0.05+random.random()*0.1,y+0.05+random.random()*0.1
```

```
            elif m.name=='Methane':
                atoms.append([x+0.1,y+0.1,z+0.1])
                atoms.append([x+0.1,y,z])
                atoms.append([x,y+0.1,z])
                atoms.append([x,y,z+0.1])
                atoms.append([x+0.1,y+0.1,z])
            elif m.name=='Ethanol':
                atoms.append([x+0.05+random.random()*0.1,y+0.05+random.random()*0.0
                atoms.append([x+0.15+random.random()*0.1,y+0.05+random.random()*0.1
                atoms.append([x+0.25+random.random()*0.1,y+0.05+random.random()*0.1
                atoms.append([x+0.05+random.random()*0.1,y+0.0,z])
                atoms.append([x+0.15+random.random()*0.1,y+0.0,z])
                atoms.append([x+0.25+random.random()*0.1,y+0.0,z])
                atoms.append([x+0.05+random.random()*0.1,y+0.2,z])
                atoms.append([x+0.25+random.random()*0.1,y+0.2,z])
                atoms.append([x+0.25+random.random()*0.1,y+0.2,z])
            else:
                print('Molecule not found ERROR')

        return np.array(atoms, dtype=float)

    ########################################################################
    def getKinEnergy(top, xyzarray, velocities):
        E_kin = np.sum(0.5*top.getMasses()*np.linalg.norm(velocities,axis=1)**2)
        return E_kin

    ########################################################################
    def getPotEnergy(top, xyzarray, bbox, cutoff):
        E_bond = 2*np.sum((calcBondPotential_MultipleAtoms(top, xyzarray)))
        E_angle = 2*np.sum((calcAnglePotential_MultipleAtoms(top,xyzarray)))
        E_LJ=np.sum((calcLennardJonesInteraction(top,xyzarray, bbox, cutoff)))
        if top.hasDiheadrals():
            #print('HAS')
```

```
        E_dih=np.sum((calcDiheadralPotential_MultipleAtoms(top,xyzarray)))
    else:
        E_dih=0.0
    E_pot = E_bond+E_angle+E_LJ+E_dih

    return E_pot

#########################################################################
def getEnergies(top, xyzarray, velocities, bbox, cutoff):
    #kinetic energy
    E_kin = getKinEnergy(top, xyzarray, velocities)
    #potential energy
    E_pot = getPotEnergy(top, xyzarray, bbox, cutoff)

    #total energy
    E_total = E_kin + E_pot

    return (E_total, E_kin, E_pot)

#########################################################################
def getShellVolume(r1,r2):
    return (4/3)*np.pi*(r2**3)-(4/3)*np.pi*(r1**3)

#########################################################################
def printEnergyHistories(top, nr_timesteps, total, kin, pot):

    plt.ylabel('kj/mol')
    plt.xlabel('time')
    plt.plot(total,label='total energy')
    plt.plot(kin,label='kinetic energy')
    plt.plot(pot,label='potential energy')

    plt.title('Energy over time')
```

```
    plt.legend()
    plt.show()

    return

#########################################################################
def printRadialDistribution(top, boxsize, nr_timesteps,
                            rdf1, rdf2, rdf3, rdf4):
    plt.subplots_adjust(hspace=0.5)
    plt.subplot(211)
    plt.plot(rdf1,label='t=0')
    plt.ylabel('g(r)')
    plt.xlabel('r in nm')
    plt.plot(rdf2,label='t='+str(nr_timesteps))
    plt.title('O-H radial distribution function')
    plt.legend()
    plt.xticks(np.arange(0,110,10),np.round(np.arange(0,boxsize/2+boxsize/20,bo

    plt.subplot(212)
    plt.ylabel('g(r)')
    plt.xlabel('r in nm')
    plt.plot(rdf3,label='t=0')
    plt.plot(rdf4,label='t='+str(nr_timesteps))

    plt.title('O-O radial distribution function')
    plt.legend()
    plt.xticks(np.arange(0,110,10),np.round(np.arange(0,boxsize/2+boxsize/20,bo
    plt.show()
    return

#########################################################################
def getRadialDistribution(xyzarray,atom1,atom2,
                          boxsize,numSamples,dr,top):
```

```
    vectors = xyzarray - xyzarray[:, np.newaxis]
    vectors = applyVectorPBC(vectors,[boxsize,boxsize,boxsize])

    distances = calcDistance(vectors, len(top.getElements()))
    atom1Indices=top.getAllAtomIndices(atom1)
    atom2Indices=top.getAllAtomIndices(atom2)
    radiusStepSize=(boxsize/2 - dr)/(numSamples+1)
    samples=np.empty(numSamples)
    samples.fill(0)

    for i in atom1Indices:
        selectedDistances=distances[i][atom2Indices]
        selectedDistances=np.sort(selectedDistances)
        for j in range(0,numSamples):
            r1=radiusStepSize*j
            r2=r1+dr
            _and=np.logical_and(selectedDistances>r1, selectedDistances<=r2)
            num=np.sum(_and)
            samples[j]+=num/getShellVolume(r1,r2)
    samples/=(len(atom1Indices)*numSamples*len(atom2Indices))
    samples*=boxsize**3*100
    return samples

##########################################################################
def runMDSimulation(topology_filename, integrator, timestep, nr_timesteps,
            cutoff, boxsize, margin, targettemp):
    top=Topology()
    top.ReadTopologyFile(topology_filename)
    xyzarray=generateUniformAtoms([boxsize,boxsize,boxsize],margin,top)
    #xyzarray=generateRandomAtoms([boxsize,boxsize,boxsize],margin,top)
    r_rdf=0.2
    rdf1=getRadialDistribution(xyzarray,'O','H',boxsize,100,r_rdf,top)
    rdf3=getRadialDistribution(xyzarray,'O','O',boxsize,100,r_rdf,top)
```

```
    if integrator=="Euler":
        numIntegrateEuler(top, xyzarray, timestep, nr_timesteps,
                cutoff, targettemp)
    elif integrator =="VelocityVerlet":
        numIntegrateVelocityVerlet(top, xyzarray, timestep,
                nr_timesteps, cutoff, targettemp)
    else:
        print("Integrator not found! ERROR!")


    rdf2=getRadialDistribution(xyzarray,'O','H',boxsize,100,r_rdf,top)
    rdf4=getRadialDistribution(xyzarray,'O','O',boxsize,100,r_rdf,top)
    printRadialDistribution(top,boxsize,nr_timesteps,rdf1,rdf2,rdf3,rdf4)
    return
#main
#########################################################################

runMDSimulation("Water_Ethanol.xml", "VelocityVerlet", 0.0002, 50, 0.8, 3, 0.0,

#########################################################################
```

6.2 Topology Code

```
import numpy as np
import lxml.etree as lxml
import sys
import matplotlib.pyplot as plt


class Molecule:

    def __init__(self, name, molid, firstatomoffset):
```

```
        self.name = name
        self.id = molid
        self.firstatomoffset = firstatomoffset
        self.atoms = []
        self.atommasses = []
        self.atomindeces = []
        self.bond_i = []
        self.bond_j = []
        self.bond_r0 = []
        self.bond_k = []
        self.angle_i = []
        self.angle_j = []
        self.angle_k = []
        self.angle_k0 = []
        self.angle_theta0 = []
        self.atom_sigma = []
        self.atom_epsilon = []
        self.diheadral_i = []
        self.diheadral_j = []
        self.diheadral_k = []
        self.diheadral_l = []
        self.diheadral_C1 = []
        self.diheadral_C2 = []
        self.diheadral_C3 = []
        self.diheadral_C4 = []

    def size(self):
        return len(self.atoms)

    def AddAtoms(self, xmlatoms):
        for atom in xmlatoms.iter('atom'):
            self.atoms.append(atom.get("type"))
            self.atommasses.append(float(atom.get("mass")))
```

```
            atomindex = int(atom.get("id"))+self.firstatomoffset
            if self.atomindeces and atomindex <= max(self.atomindeces):
                print(
                    "Atoms␣in␣the␣topology␣section␣have␣to␣be␣sorted␣according␣
                sys.exit(1)
            self.atomindeces.append(atomindex)
            self.atom_sigma.append(float(atom.get("sigma")))
            self.atom_epsilon.append(float(atom.get("epsilon")))

    def AddBonds(self, xmlbonds):
        for bond in xmlbonds.iter('bond'):
            self.bond_i.append(int(bond.get("i"))+self.firstatomoffset)
            self.bond_j.append(int(bond.get("j"))+self.firstatomoffset)
            self.bond_r0.append(float(bond.get("r0")))
            self.bond_k.append(float(bond.get("k")))

    def AddAngles(self, xmlangles):
        for angle in xmlangles.iter('angle'):
            self.angle_i.append(int(angle.get("i"))+self.firstatomoffset)
            self.angle_j.append(int(angle.get("j"))+self.firstatomoffset)
            self.angle_k.append(int(angle.get("k"))+self.firstatomoffset)
            self.angle_k0.append(float(angle.get("k0")))
            self.angle_theta0.append(float(angle.get("theta0")))

    def AddDiheadrals(self, xmldiheadrals):
        for diheadral in xmldiheadrals.iter('diheadral'):
            self.diheadral_i.append(int(diheadral.get("i"))
            +self.firstatomoffset)
            self.diheadral_j.append(int(diheadral.get("j"))
            +self.firstatomoffset)
            self.diheadral_k.append(int(diheadral.get("k"))
            +self.firstatomoffset)
            self.diheadral_l.append(int(diheadral.get("l"))
```

```
                +self.firstatomoffset)
                self.diheadral_C1.append(float(diheadral.get("C1")))
                self.diheadral_C2.append(float(diheadral.get("C2")))
                self.diheadral_C3.append(float(diheadral.get("C3")))
                self.diheadral_C4.append(float(diheadral.get("C4")))

        def print(self):
            print("Name:{} Id:{} Offset:{} Size:{} Bonds:{} Angles:{} ".format(
                self.name, self.id, self.firstatomoffset, self.size(),
                len(self.bond_i), len(self.angle_i)))


class Topology:

    def __init__(self):
        self.molnames = []
        self.molecules = []
        self.box = 0.0

    def getLennardJonesMatrix2d(self):
        m = np.full((self.getNumOfAtoms(),self.getNumOfAtoms()),1.0)
        for mol in self.molecules:
            atoms=mol.atomindeces
            first=atoms[0]
            last=atoms[len(atoms)-1]+1
            m[first:last,first:last]*=0.0
        return m

    def getLennardJonesMatrix(self):
        m = np.full((self.getNumOfAtoms(),self.getNumOfAtoms(),3),1.0)
        for mol in self.molecules:
            atoms=mol.atomindeces
            first=atoms[0]
```

```
            last=atoms[len(atoms)-1]+1
            m[first:last,first:last]*=0.0
        return m

    def printAtoms(self):
        for mol in self.molecules:
            print(mol.atomindeces)
        return

    # get indices of all atoms of a certain type (e.g. 'H')
    def getAllAtomIndices(self,name):
        ind=[]
        count=0
        for mol in self.molecules:
            for atom in mol.atoms:
                if(atom==name):
                    ind.append(count)
                count+=1
        return ind

    def hasDiheadrals(self):
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.diheadral_i)
        return len(indeces)>0

    def getNumOfMolecules(self):
        return len(self.molecules)

    def getNumofmolnames(self):
        return len(self.molnames)

    def getElements(self):
```

```
        elements = []
        for mol in self.molecules:
            elements.extend(mol.atoms)
        return elements

    def getNumOfAtoms(self):
        elements = []
        for mol in self.molecules:
            elements.extend(mol.atoms)
        return len(elements)

    def getMasses(self):
        masses = []
        for mol in self.molecules:
            masses.extend(mol.atommasses)
        return np.array(masses)

    def getAtomIndeces(self, molid):
        return np.array(self.molecules[molid].atomindeces)

    def getBond_i_indeces(self):
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.bond_i)
        return np.array(indeces)

    def getBond_j_indeces(self):
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.bond_j)
        return np.array(indeces)

    def getBond_r0(self):
```

```
        r0 = []
        for mol in self.molecules:
            r0.extend(mol.bond_r0)
        return np.array(r0)

    def getBond_k(self):
        k = []
        for mol in self.molecules:
            k.extend(mol.bond_k)
        return np.array(k)

    def getAngle_i_indeces(self):
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.angle_i)
        return np.array(indeces)

    def getAngle_j_indeces(self):
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.angle_j)
        return np.array(indeces)

    def getAngle_k_indeces(self):
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.angle_k)
        return np.array(indeces)

    def getAngle_k0(self):
        k0 = []
        for mol in self.molecules:
            k0.extend(mol.angle_k0)
```

```
        return np.array(k0)

    def getAngle_theta0(self):
        theta0 = []
        for mol in self.molecules:
            theta0.extend(mol.angle_theta0)
        return np.array(theta0)

    def getAtom_Sigma(self):
        sigma = []
        for mol in self.molecules:
            sigma.extend(mol.atom_sigma)
        return np.array(sigma)

    def getAtom_Epsilon(self):
        epsilon = []
        for mol in self.molecules:
            epsilon.extend(mol.atom_epsilon)
        return np.array(epsilon)

    def getDiheadral_i_indeces(self):
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.diheadral_i)
        return np.array(indeces)

    def getDiheadral_j_indeces(self):
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.diheadral_j)
        return np.array(indeces)

    def getDiheadral_k_indeces(self):
```

```
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.diheadral_k)
        return np.array(indeces)

    def getDiheadral_l_indeces(self):
        indeces = []
        for mol in self.molecules:
            indeces.extend(mol.diheadral_l)
        return np.array(indeces)

    def getDiheadral_C1(self):
        C1 = []
        for mol in self.molecules:
            C1.extend(mol.diheadral_C1)
        return np.array(C1)

    def getDiheadral_C2(self):
        C2 = []
        for mol in self.molecules:
            C2.extend(mol.diheadral_C2)
        return np.array(C2)

    def getDiheadral_C3(self):
        C3 = []
        for mol in self.molecules:
            C3.extend(mol.diheadral_C3)
        return np.array(C3)

    def getDiheadral_C4(self):
        C4 = []
        for mol in self.molecules:
            C4.extend(mol.diheadral_C4)
```

```
        return np.array(C4)

    def getMolecules(self):
        return self.molecules

    def ReadTopologyFile(self, topoFile):
        print("Parsing  {}....".format(topoFile))
        tree = lxml.parse(topoFile)
        root = tree.getroot()
        self.box = float(root.find("box").text)
        atomoffset = 0
        for mol in root.find("molecules").iter('molecule'):
            molname = mol.get("name")
            if(molname in self.molnames):
                print("Defined molecule with name {} twice.".format(molname))
                sys.exit(1)
            else:
                self.molnames.append(molname)
            atomoffset = self.AddMolecule(mol, atomoffset)

    def AddMolecule(self, xmlentry, atomoffset):
        number_of_mol = int(xmlentry.get("number"))
        molname = xmlentry.get("name")
        if int(xmlentry.get("id")) != len(self.molnames)-1:
            print(
                "Molecule ids must start from zero and the first molecule must
            sys.exit(1)
        lastmolID = len(self.molecules)
        for molid in range(number_of_mol):
            mol = Molecule(molname, molid+lastmolID, atomoffset)
            mol.AddAtoms(xmlentry.find("atoms"))
            mol.AddBonds(xmlentry.find("bonds"))
            mol.AddAngles(xmlentry.find("angles"))
```

```
                mol.AddDiheadrals(xmlentry.find('diheadrals'))
                self.molecules.append(mol)
                atomoffset += mol.size()
            print("Added {} molecules of name {}".format(number_of_mol, molname))
            return atomoffset

def ReadElementsFromTrajectory(trajFile):
    """This will read the first frame of the trajectory into memory."""
    trajectory = []
    with open(trajFile, "r") as tF:
        line = tF.readline()
        # first line is number of atoms
        N = int(line.strip())
        tF.readline().strip()  # second line is a comment that we throw away
        element = []
        for i in range(N):
            line = tF.readline().strip().split(" ")
            element.append(line[0])
    return element


def verifyTrajectory(trajfile, topology):
    traj_elements = ReadElementsFromTrajectory(trajfile)
    elements = topology.getElements()
    return traj_elements == elements


def ReadTrajectory(trajFile):
    trajectory = []
    with open(trajFile, "r") as tF:
        line = tF.readline()
        while line:
            # first line is number of atoms
```

```
            N = int(line.strip())
            tF.readline().strip()

            q = []
            for i in range(N):
                line = tF.readline().strip().split(" ")
                for c in line[1:]:
                    if c is not "":
                        q.append(float(c))
            trajectory.append(np.array(q))

            line = tF.readline()

    return trajectory, N
```

6.3 Topology XML File

```
<topology>
    <box>91</box>
    <molecules>
        <molecule name="Water" id="0" number="200">
            <atoms>
                <atom type="O" id="0" mass="16" sigma="0.315061" epsilon="0.66386"/>
                <atom type="H" id="1" mass="1" sigma="0.00" epsilon="0.00"/>
                <atom type="H" id="2" mass="1" sigma="0.00" epsilon="0.00"/>
            </atoms>
            <bonds>
                <bond i="0" j="2" r0="0.09572" k="502416"/>
                <bond i="0" j="1" r0="0.09572" k="502416"/>
            </bonds>
            <angles>
                <angle i="1" j="0" k="2" k0="628.02" theta0="1.82"/>
            </angles>
            <diheadrals>
            </diheadrals>
```

```
</molecule>
<molecule name="Ethanol" id="1" number="33">
    <atoms>
        <atom type="C" id="0" mass="12" sigma="0.35" epsilon="0.276144"/>
        <atom type="C" id="1" mass="12" sigma="0.35" epsilon="0.276144"/>
        <atom type="O" id="2" mass="16" sigma="0.312" epsilon="0.711280"/>
        <atom type="H" id="3" mass="1" sigma="0.0" epsilon="0.0"/>
        <atom type="H" id="4" mass="1" sigma="0.0" epsilon="0.0"/>
        <atom type="H" id="5" mass="1" sigma="0.0" epsilon="0.0"/>
        <atom type="H" id="6" mass="1" sigma="0.0" epsilon="0.0"/>
        <atom type="H" id="7" mass="1" sigma="0.0" epsilon="0.0"/>
        <atom type="H" id="8" mass="1" sigma="0.25" epsilon="0.125520"/>
    </atoms>
    <bonds>
        <bond i="0" j="3" r0="0.1090" k="284512.0"/>
        <bond i="0" j="4" r0="0.1090" k="284512.0"/>
        <bond i="0" j="5" r0="0.1090" k="284512.0"/>
        <bond i="1" j="6" r0="0.1090" k="284512.0"/>
        <bond i="1" j="7" r0="0.1090" k="284512.0"/>
        <bond i="2" j="8" r0="0.0945" k="462750.0"/>
        <bond i="0" j="1" r0="0.1529" k="224262.4"/>
        <bond i="1" j="2" r0="0.1410" k="267776.0"/>
    </bonds>
    <angles>
        <angle i="3" j="0" k="1" k0="292.880" theta0="1.894"/>
        <angle i="4" j="0" k="1" k0="292.880" theta0="1.894"/>
        <angle i="5" j="0" k="1" k0="292.880" theta0="1.894"/>
        <angle i="3" j="0" k="4" k0="276.144" theta0="1.881"/>
        <angle i="3" j="0" k="5" k0="276.144" theta0="1.881"/>
        <angle i="4" j="0" k="5" k0="276.144" theta0="1.881"/>
        <angle i="6" j="1" k="7" k0="276.144" theta0="1.881"/>
        <angle i="0" j="1" k="6" k0="313.800" theta0="1.932"/>
        <angle i="0" j="1" k="7" k0="313.800" theta0="1.932"/>
        <angle i="0" j="1" k="2" k0="414.400" theta0="1.911"/>
        <angle i="1" j="2" k="8" k0="460.240" theta0="1.894"/>
        <angle i="6" j="1" k="2" k0="292.880" theta0="1.911"/>
```

```
            <angle i="7" j="1" k="2" k0="292.880" theta0="1.911"/>
        </angles>
        <diheadrals>
            <diheadral i="3" j="0" k="1" l="6" C1="0.62760" C2="1.88280"
                C3="0.00" C4="-3.91622"/>
            <diheadral i="4" j="0" k="1" l="6" C1="0.62760" C2="1.88280"
                C3="0.00" C4="-3.91622"/>
            <diheadral i="5" j="0" k="1" l="6" C1="0.62760" C2="1.88280"
                C3="0.00" C4="-3.91622"/>
            <diheadral i="3" j="0" k="1" l="7" C1="0.62760" C2="1.88280"
                C3="0.00" C4="-3.91622"/>
            <diheadral i="4" j="0" k="1" l="7" C1="0.62760" C2="1.88280"
                C3="0.00" C4="-3.91622"/>
            <diheadral i="5" j="0" k="1" l="7" C1="0.62760" C2="1.88280"
                C3="0.00" C4="-3.91622"/>
            <diheadral i="3" j="0" k="1" l="2" C1="0.97905" C2="2.93716"
                C3="0.00" C4="-3.91622"/>
            <diheadral i="4" j="0" k="1" l="2" C1="0.97905" C2="2.93716"
                C3="0.00" C4="-3.91622"/>
            <diheadral i="5" j="0" k="1" l="2" C1="0.97905" C2="2.93716"
                C3="0.00" C4="-3.91622"/>
            <diheadral i="0" j="1" k="2" l="8" C1="-0.44310"

                C2="3.83255" C3="0.72801" C4="-4.11705"/>
            <diheadral i="6" j="1" k="2" l="8" C1="0.94140" C2="2.82420"
                C3="0.00" C4="-3.76560"/>
            <diheadral i="7" j="1" k="2" l="8" C1="0.94140" C2="2.82420"
                C3="0.00" C4="-3.76560"/>
        </diheadrals>
    </molecule>
</molecules>
</topology>
```

7 References

References

[1] I Juurinen et al. “Measurement of Two Solvation Regimes in Water-Ethanol Mixtures Using X-Ray Compton Scattering”. In: *Physical review letters* 107 (Nov. 2011), p. 197401. DOI: 10.1103/PhysRevLett.107.197401.

[2] M. Mijaković et al. “Ethanol-water mixtures: ultrasonics, Brillouin scattering and molecular dynamics”. In: *Journal of Molecular Liquids* 164.1 (2011). Complex liquids: Modern trends in exploration, understanding and application Selected papers on molecular liquids presented at the EMLG/JMLG annual meeting 5-9 september 2010, pp. 66–73. ISSN: 0167-7322. DOI: https://doi.org/10.1016/j.molliq.2011.06.009. URL: http://www.sciencedirect.com/science/article/pii/S0167732211002145.

[3] Nobuyuki. Nishi et al. “Molecular association in ethanol-water mixtures studied by mass spectrometric analysis of clusters generated through adiabatic expansion of liquid jets”. In: *Journal of the American Chemical Society* 110.16 (1988), pp. 5246–5255. DOI: 10.1021/ja00224a002. eprint: https://doi.org/10.1021/ja00224a002. URL: https://doi.org/10.1021/ja00224a002.

[4] Katsuko Takaizumi. “A Curious Phenomenon in the Freezing–Thawing Process of Aqueous Ethanol Solution”. In: *Journal of Solution Chemistry* 34.5 (May 2005), pp. 597–612. ISSN: 1572-8927. DOI: 10.1007/s10953-005-5595-6. URL: https://doi.org/10.1007/s10953-005-5595-6.

Figure 1: https:\\www.quora.com/What-molecule-is-present-in-water

Figure 2: https:\\www.researchgate.net/figure/Isolated-ethanol-molecule_fig2_252766155