Looking for possible hydrogen bonds and water clustering in a microemulsion system

Maria Minakova

A microemulsion system consists of three types of molecules: water, surfactant and oil. This particular system was studied at all-atom resolution in a rectangular simulation box with Periodic Boundary Conditions (PBC). Visual analysis showed that water does manage to go through the oil layer, though it is highly energetically costly. There are 2 plausible mechanisms of water transport: hydrated surfactants and micelles. Hydrated surfactants represent disordered individual or multiple surfactant molecules that weakly bind one or several water molecules by their "heads". This effectively shields water molecules from unfavorable interactions with oil. The "Hydrated surfactants" type of transport assumes that individual or small disordered groups of surfactants diffuse through the oil layer and transport water with them. The micellar transport requires system self-organization into water droplets (micelle core), surrounded by surfactants (micelle corona). To illuminate the way water travels through oil in this chemical mixture, I needed to:

1. Identify possible interactions that can lead to self-organization. In absence of other influential factors, like ions and salts, water and surfactant molecules can bind weakly via hydrogen bonding. Thus make lists of all possible combinations between oxygens and hydrogens of surfactant heads and water.

2. Compute radial distribution functions (RDF) for all possible pairwise atom/residue combinations, look at the most likely distance, average distance, first solvation shell (location of the 1st minimum after the 1st maximum of the RDF). Use the average solvation shell size for most prominent binding pairs as a cluster linkage parameter.

3. While calculating RDF, fill out lists of neighbors for each species of interest based on linking parameter values. Note: water-water pairs have a different linkage parameter, than water-surfactant and surfactant-surfactant pairs.

4. Use iterative percolation procedure to segregate all water molecules into clusters. Water-water cluster probability distribution depends on the number of water molecules in it $P_{ww}(N)$ and is thus 1D. Water-surfactant clustering also depends on the number of surfactants M per cluster $P_{ws}(N, M)$ and is thus 2D. Marginal probability $P_{ws}(N) = \sum_{M=1}^{M_{total}} P(N, M)$ should be compared with pure water clusters $P_{ww}(N)$. The shape of $P_{ww}(N)$ tells us about direct water contacts, hence its self-organization (micelle cores?), while the $P_{ws}(N)$ can reveal the presence large disordered water clusters, where water molecules are connected through surfactants (hydrated surfactants?).

Here is what the simulation system looks like:

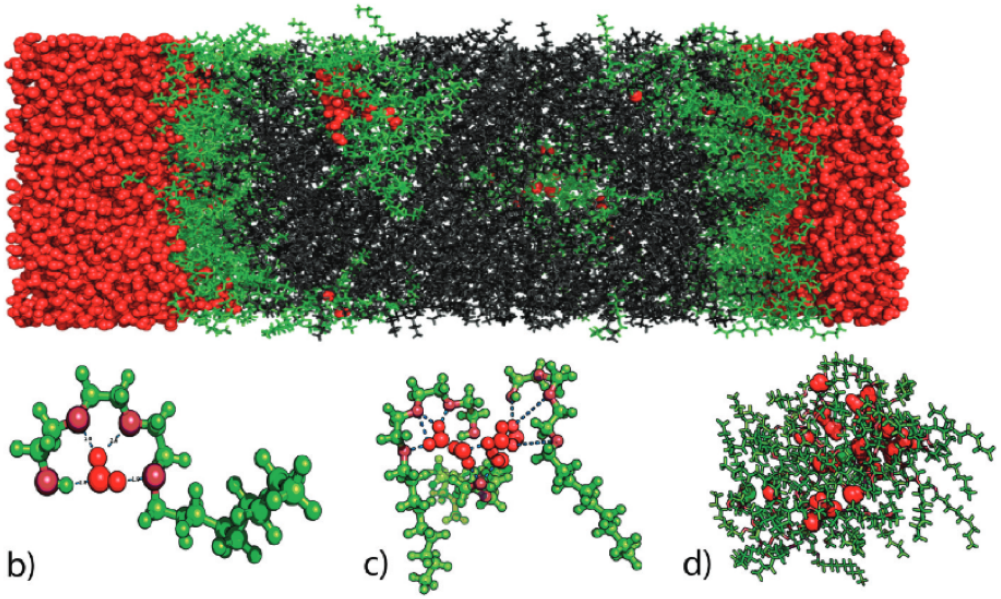

Figure 1. (a) A snapshot of the system in a box at $T_1 = 25$ °C. Color scheme: water molecules are red, surfactant molecules are green, and oil molecules are gray. The water−surfactant complexes formed at the boundary of/in the oil layer can be seen. (b) A snapshot of a single surfactant molecule solvating one water molecule. Color scheme: green spheres are surfactant carbons and hydrogens, ruby spheres are surfactant oxygens, and red spheres are water hydrogens and oxygens. (c) A snapshot of several surfactant molecules solvating several water molecules. The color scheme is the same as that in (b). (d) A snapshot of a micellar-like aggregate, including surfactant molecules (green) and water (red). Oil molecules in (b−d) are removed for the lucid demonstration of the water−surfactant complexes.

And these are the resulting probability distributions $P_{ww}(N)$ and $P_{ws}(N)$:

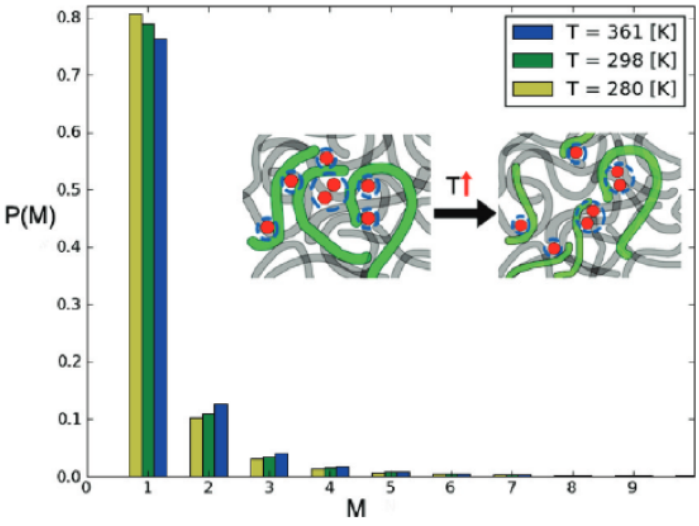

Figure 2. Probability distribution function $P(M)$ for 1D clustering of water molecules shown for several temperatures T = 7, 25, and 88 °C. The most probable cluster size is given by the maximum of $P(M)$ and is equal to 1 for all temperatures. With the temperature increase, the probability distribution becomes wider, suggesting that larger clusters are permitted in the hydrophobic region. This effect is partially due to the overall increase of the water density on the oil slab. The change in the clusters' distribution with temperature provided by the visual analysis is schematically shown in the inset. Color scheme: water molecules are red, surfactant molecules are green, and oil molecules are gray. Note the difference between the clusters obtained from 1D and 2D clustering (see Figure 3).

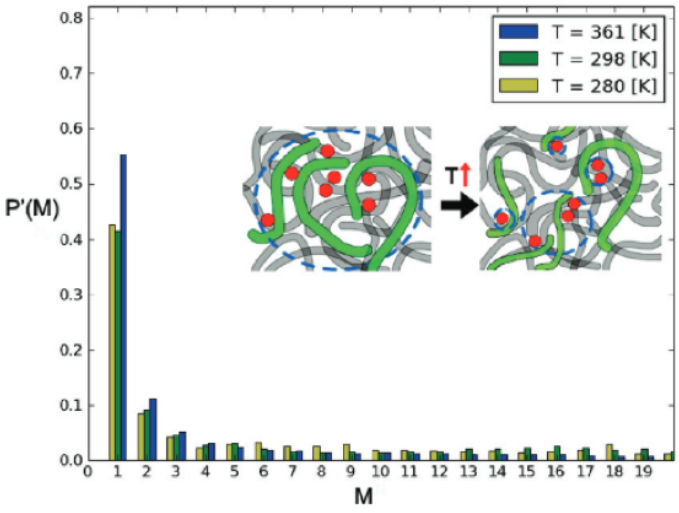

Figure 3. Marginal probability distribution function $P'(M)$ for 2D clustering of water molecules shown for several temperatures T = 7, 25, and 88 °C. The most probable cluster size is given by the maximum of $P'(M)$ and is equal to 1 for all temperatures. With the temperature increase, the probability distribution becomes narrower, suggesting that clusters dissociate into smaller ones in the hydrophobic region. The change in the clusters' distribution with temperature provided by the visual analysis is schematically shown in the inset. Color scheme: water molecules are red, surfactant molecules are green, and oil molecules are gray. Note the difference between the clusters obtained from 1D and 2D clustering (see Figure 2).

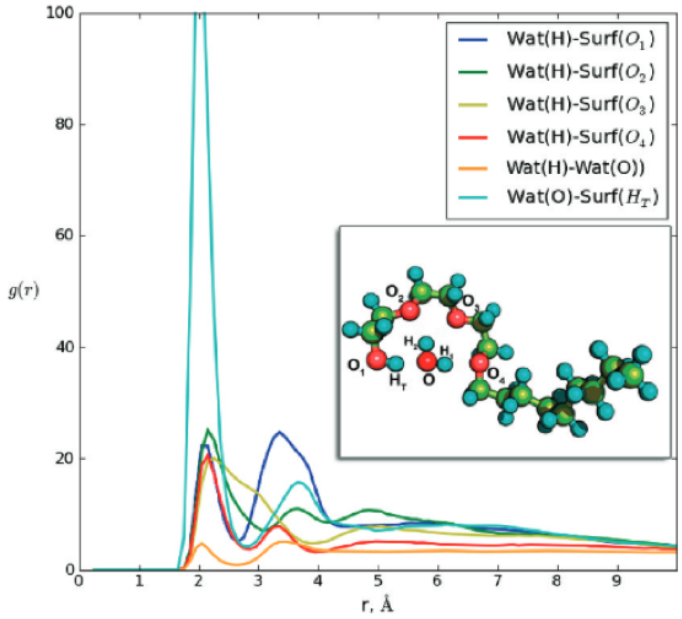

Figure 4. RDF of various atom pairs of water and surfactant that can potentially contribute to hydrogen bond formation in the oil slab. Because our main interest is the water−surfactant complexes in oil, these RDFs has been computed for the molecules located in the oil layer. The notations for RDF atom pairs are shown in the inset, which is a snapshot of a water−surfactant complex taken from the simulations.

I ran simulations on an HPC cluster in a parallelized fashion. For each external parameter value (temperature) I had created its own simulation system, and systems could swap configurations through Monte Carlo sampling technique, called "replica exchange". Therefore the analysis code shown below had to run for each ext. parameter value. I wrote nested submission scripts in Bash to automate that process and ran the executable of the C++ code below on a HPC Killdevil cluster, because all-atom trajectory files are too big to attempt parsing and analyzing them on a single CPU. Code below shows a main part of the C++ program that opens files that contain all-atom information about a microemulsion system, using specialized C++ library BALL. These files are called DCDs and contain terabyte scale semi-structured data about all atoms, saved for many time frames (windows). For each file program calls the data analysis functions from "snapshot.cpp" and uses "averages.cpp" to write down the results for plotting and further analysis.

Scipy and Matplotlib Python libraries were used to plot the results, and C++ code that provided the final data for plotting is below (shortened, only relevant clustering methods are unfolded).

1 Main.cpp

```
#include <iostream>
#include <list>
#include <vector>
#include <cmath>
#include <fstream>
#include <string.h>
#include <cstdlib>
#include <new>
#include <boost/lexical_cast.hpp>
#include <stdio.h>
// the BALL kernel classes
#include <BALL/KERNEL/atom.h>
#include <BALL/KERNEL/residue.h>
#include <BALL/KERNEL/system.h>
#include <BALL/KERNEL/PTE.h>
#include <BALL/KERNEL/selector.h>
#include <BALL/KERNEL/bond.h>
#include <BALL/STRUCTURE/fragmentDB.h>
#include <BALL/STRUCTURE/residueChecker.h>
#include <BALL/MATHS/vector3.h>
#include <BALL/MATHS/analyticalGeometry.h>
#include <BALL/MATHS/angle.h>
// reading and writing of PDB files
#include <BALL/FORMAT/PDBFile.h>
#include <BALL/FORMAT/DCDFile.h>
#include <BALL/MOLMEC/COMMON/snapShotManager.h>
#include <BALL/FORMAT/PDBFile.h>
#include <BALL/STRUCTURE/geometricProperties.h>
#include <BALL/STRUCTURE/geometricTransformations.h>
//My header files
#include "snapshot.h"
#include "averages.h"
using namespace BALL;
using namespace std;
using namespace boost;
int main()
{//                                  Declarations                                    //
        string inputfolder;                 //folder manipulator
        string inputfname;                  //file mask without extension
        ofstream myoutfile;
        //int firstfile = 4;                            //index of 1st file
        int firstfile = 15;
        //int numfiles = 12;                            //number of DCD windows
        int numfiles = 1;
        System S1;                          //BALL class object that references the full system
        int n_tot=0;                        //total number of frames in simulation.
        int windows;                        //time window counter for *.dcd trajectory files
        bool firstpass=0;                   //flag that all residues were found
        int miwatnum;                       //water residue counter;
        double thickness=8;                 //thickness of the surfactant layer in Angstroms
        double MaxValue = 20;               //for Radial Dist. - max distance for data histograming
        int ClustNum=80;                    //Max number of clusters. for water clustering (just
            ↪ some number large enough).
        bool ClAnalysflag = 0;              //marker - whether 2D clustering was performed for the
            ↪ snapshot (within snapshot loop).
```

```
        double K1,K2, Kwatave, Ksurfave;    //partition coefficient.
        ResidueIterator res_it;             //residue iterator
        ResidueIterator res_jt;
        Selector sel_water("residue(WAT)"); //Selector expression to find all water molecules(
            ↪ residues)
        Selector sel_ant("residue(ant)");   //Selector expression to find all surfactant molecules(
            ↪ residues)
        Selector sel_head("residue(ant) AND name(H24) OR name(O23) OR name(H55) OR name(H54) OR name
            ↪ (C22)");
        AtomIterator at_it;                 // Atom iterator handle to use on surfactant molecule
            ↪ objects
        Snapshot* OneSnap;                  //class with handles and clustering methods for a single
            ↪  instance of the system (one time frame)
        Averages* MainAve = new Averages;   //where to store computed values and averages of all
            ↪ kinds
        Hist* CurrentHist;                  //Histogram handle
        int RDF_count = 1;                  //how many radial distribution functions will be
            ↪ calculated
        MainAve->allRDs = new Hist[RDF_count];
        MainAve->Ksurf = 0;
        MainAve->Kwater = 0;
        MainAve->framecount = 0;
        MainAve->clustframenum=0;
        OneSnap = new Snapshot;
        OneSnap->framecount = 0;
        OneSnap->filecount=numfiles;
//              Open parameter file, structure file, check the simulation box size.
        inputfolder="/home/usr/codes/microem/data/280K/";
        inputfname = "repl42_";
        string pdbname = inputfolder + "all.pdb";
        PDBFile input2(pdbname);
        cout << "\nMain cycle: PDB file is : " << pdbname << " is uploaded." << endl;
        input2 >> S1;
        input2.close();
        OneSnap->S1=&S1;
        string box_file_name = inputfolder + inputfname + ".box";
        cout<<" Main Cycle: Box size is going to be read from : "<<box_file_name<<endl;
        OneSnap->SetBoxSize(box_file_name, 1); //1 - NVT ensemble, 0 - NPT ensemble.
//                          Cycle through runs
    for(windows=firstfile; windows<numfiles+firstfile; windows++)
    {//              open file                   //
        string traj_file_name = inputfolder + inputfname + lexical_cast<string>(windows) + "_temp280
            ↪ .dcd";
        cout <<"Main cycle: Reading " << traj_file_name << endl;
        DCDFile dcd(traj_file_name);
        SnapShot ssh;
        int gen_skip = 1; //how many frames skip before reading another one.
        int n_frames=dcd.getNumberOfSnapShots();
        OneSnap->framecount+= static_cast<int>(n_frames/gen_skip);//convert: 14.9978 -> 14,
            ↪ 14.00001-> 14.
        MainAve->framecount+= static_cast<int>(n_frames/gen_skip);
        cout<<"Main cycle: OneSnap->framecount is increased by "<<static_cast<int>(n_frames/gen_skip
            ↪ )<<"  frames and = "<<OneSnap->framecount<<endl;
//                      Cycle through snapshots for each run                    //
        for(int n_f=0; n_f<n_frames; ++n_f)
        {
                dcd.read(ssh);
                ssh.applySnapShot(S1);
                if(n_f%gen_skip!=0)
                continue;
                if (n_tot % 1==0)                   //       this section
                {                                     //     displays progress
                cout << "\n************FRAME: " << n_tot << "******************\n"; //          of
                    ↪ the
                }                                     //          program
//          Initialization of the object OneSnap of the class snapshot
                if (n_f!=0)
                {
                        firstpass=1;
                }
                else
                {
```

```
                        OneSnap->setResNumbers(firstpass, MainAve);
                }
                OneSnap->Phi_per_slab(firstpass, MainAve, n_f,1, 1); // calculate mass distribution through the oil slab (Z axis)
                OneSnap->Phi_per_slab(firstpass, MainAve, n_f,1, 0); // calculate mass distribution through the oil slab (Z axis)
//          Water 1D clustering
        CurrentHist = &MainAve->allRDs[0];
                bool RD_successOH = OneSnap->RadialDist(n_f, CurrentHist, firstpass, 2, "residue(WAT) AND (name(H1) OR name(H2))", "residue(WAT) AND name(O)", "WatH_WatO_all_", MaxValue, 1);          // oxygen - hydrogen (hydrogen bonds)
                bool RD_successOO = OneSnap->RadialDist(n_f, CurrentHist, firstpass, 2, "residue(WAT) AND name(O)", "residue(WAT) AND name(O)", "WatO_WatO_all_", MaxValue, 1);
                                        // oxygen - oxygen distance distribution
        OneSnap->Clustering(MainAve, firstpass, ClustNum);
//          Calculation of Radial Distribution functions of all meaningful atom-atom cross molecule interactions, looking for weak binding
                CurrentHist = &MainAve->allRDs[0];
                bool RD_success1 = OneSnap->AtomRadialDist_in_Slab(n_f, CurrentHist, firstpass, "residue(ant) AND name(H24)", "residue(WAT) AND name(O)", "0_SurfH24_WatO_oilsurf_", MaxValue, 1);

                CurrentHist = &MainAve->allRDs[1];
                bool RD_success2 = OneSnap->AtomRadialDist_in_Slab(n_f, CurrentHist, firstpass, "residue(ant) AND name(O23)","residue(WAT) AND (name(H1) OR name(H2))", "1_SurfO23_WatH_oilsurf_", MaxValue, 1);

                CurrentHist = &MainAve->allRDs[2];
                RD_success2 = OneSnap->AtomRadialDist_in_Slab(n_f, CurrentHist, firstpass,"residue(ant) AND name(O20)", "residue(WAT) AND (name(H2) OR name(H1))", "2_SurfO20_WatH_oilsurf_", MaxValue, 1);

                CurrentHist = &MainAve->allRDs[3];
                RD_success2 = OneSnap->AtomRadialDist_in_Slab(n_f, CurrentHist, firstpass, "residue(ant) AND name(O17)", "residue(WAT) AND (name(H2) OR name(H1))", "3_SurfO17_WatH_oilsurf_", MaxValue, 1);

                CurrentHist = &MainAve->allRDs[4];
                RD_success2 = OneSnap->AtomRadialDist_in_Slab(n_f, CurrentHist, firstpass, "residue(ant) AND name(O12)", "residue(WAT) AND (name(H2) OR name(H1))", "4_SurfO12_WatH_oilsurf_", MaxValue, 1);

                CurrentHist = &MainAve->allRDs[5];
                RD_success2 = OneSnap->AtomRadialDist_in_Slab(n_f, CurrentHist, firstpass, "residue(WAT) AND (name(H1) OR name(H2))", "residue(WAT) AND name(O)", "5_WatO_WatH_oilsurf_", MaxValue, 1);

                CurrentHist = &MainAve->allRDs[6];
                RD_success2 = OneSnap->AtomRadialDist_in_Slab(n_f, CurrentHist, firstpass, "residue(WAT) AND (name(H1) OR name(H2))", "residue(WAT) AND name(O)", "6_WatO_WatH_inwater_", MaxValue, 3);

        CurrentHist = &MainAve->allRDs[5];
                OneSnap->RadialDist(n_f, CurrentHist, firstpass, 2, "residue(WAT) AND (name(H1) OR name(H2))", "residue(WAT) AND name(O)", "7_WatH_WatO_all_", MaxValue, 0);

                CurrentHist = &MainAve->allRDs[0];
                bool RD_success1 = OneSnap->RadialDist(n_f, CurrentHist, firstpass,2, "residue(ant) AND name(H24)", "residue(WAT) AND name(O)", "8_SurfH24_WatO_all_", MaxValue, 0);

                CurrentHist = &MainAve->allRDs[1];
                bool RD_success2 = OneSnap->RadialDist(n_f, CurrentHist, firstpass,2, "residue(ant) AND name(O23)","residue(WAT) AND (name(H1) OR name(H2))", "9_SurfO23_WatH_all_", MaxValue, 0);

                CurrentHist = &MainAve->allRDs[2];
                bool RD_success3 = OneSnap->RadialDist(n_f, CurrentHist, firstpass,2, "residue(ant) AND name(O20)", "residue(WAT) AND (name(H2) OR name(H1))", "10_SurfO20_WatH_all_", MaxValue, 0);

                CurrentHist = &MainAve->allRDs[3];
```

```
                bool RD_success4 = OneSnap->RadialDist(n_f, CurrentHist, firstpass,2, "residue(ant)
                    ↪ AND name(O17)", "residue(WAT) AND (name(H2) OR name(H1))", "11
                    ↪ _SurfO17_WatH_all_", MaxValue, 0);

                CurrentHist = &MainAve->allRDs[4];
                bool RD_success5 = OneSnap->RadialDist(n_f, CurrentHist, firstpass,2, "residue(ant)
                    ↪ AND name(O12)", "residue(WAT) AND (name(H2) OR name(H1))", "12
                    ↪ _SurfO12_WatH_all_", MaxValue, 0);

                //OneSnap->printfWatObjects();  //print where all water molecules are for ploting
                    ↪ and visual analysis
//2D Clustering part:
                CurrentHist = &MainAve->allRDs[0];
                ClAnalysflag = OneSnap->WS_clustering2D(MainAve, CurrentHist,  firstpass, ClustNum);
                if (ClAnalysflag ==1)
                MainAve->clustframenum++;
                miwatnum=OneSnap->watcount;
                n_tot++;

                MainAve->NormAndWrite_btwn(RDF_count, n_tot,OneSnap->LinkParam_ww, OneSnap->
                    ↪ LinkParam_ws);
        }//cycle trough frames
    } //cycle through files
MainAve->NormAndWrite(RDF_count, OneSnap->LinkParam_ww, OneSnap->LinkParam_ws); //normalize and
    ↪ write down averages

cout<<"Number of residues in the system is = "<<miwatnum<<endl;
delete MainAve;
delete OneSnap;

cout<<"Main cycle: Program has finished.\n";
} //main
```

2 Snapshot.cpp

```
#include "snapshot.h"
#include <boost/lexical_cast.hpp>
Snapshot::Snapshot()
{}
Snapshot::~Snapshot()
{}
bool Snapshot::IsInList(list<int> listCluster,int value) //Check if a residue\atom is in the cluster
    ↪  member list
{...}

void Snapshot::setResNumbers(bool firstpass, Averages* Ave)//Assign individual IDs to atom groups
{...}

void Snapshot::createMolObjects(ParticleandLinks** Objects, string selector, bool firstpass, int
    ↪ liSize, string Type)//create containters and fill out info about water molecules
{...}

void Snapshot::deleteMolObjects(ParticleandLinks* Objects)//clean containers for water info
{...}

void Snapshot::printfWatObjects() //find and print center of masses for all water molecules
{...}

void Snapshot::Lipid_heads_Surface() //Find all atoms belonging to lipid head, find mass center and
    ↪ make a surface using them as vertices.
{...}

void Snapshot::Phi_per_slab(bool firstpass, Averages* Ave, int framecounter, bool DoSymmAnalysis,
    ↪ bool DoShift) // Calc. 1D distriubtion function of water, oil, and sufractant weight fraction
    ↪  perpendicular to layers (Z axis)
{...}

void Snapshot::SymmetryAnalysis() //Ideally our system consists of 5 layers: water, surfactant, oil,
    ↪  surfactant, water. Due to PBCs layers can shift during the simulations. If so, shift of all
    ↪ atoms around the enclosed surface is required to maintain the same structure for further
```

```
    ↪ analysis
{...}

void Snapshot::ShiftAlongNormal() //System shifted during the simulation - shift with periodic
    ↪ boundary conditions in mind to mantain 5 layer structure
{...}

bool Snapshot::RadialDist(int framenumber, Hist* CurrentHist, bool firstpass, int type, String
    ↪ nameforselect1, String nameforselect2, string outname, double MaxValue, bool doclust) //RDF
{
        cout<<"FUNCTION CALL: RadialDist for selector1 = "<<nameforselect1<<" and selector2 = "<<
            ↪ nameforselect2<<endl;
        Selector sel_obj1(nameforselect1);
        Selector sel_obj2(nameforselect2);
        double ldDist;
        double ldDensity2;
        int resicount=0;
        int resjcount=0;
        bool speciesfound=0;
        Vector3 center_of_mass_i(0,0,0);
        Vector3 center_of_mass_j(0,0,0);
        Vector3 diff_ij(0,0,0);
        Residue *res_i, *res_j;
        Atom::StaticAtomAttributes *d1,*d2;
        stringstream ss (stringstream::in | stringstream::out);
        int ID1=0, ID2=0;
//              Initialization of Radial Distribution function
        int RDsize=int(MaxValue/RDbinSize)+1;
        Array<double,1> RadialDistHist(RDsize); //Main array where the N(r) will be put.
        Array<double,1> RadialDistBins(RDsize);
        Array<double,1> Jacobian(RDsize); //Jacobian for normaqlization of the g(r)
        firstIndex i;
        RadialDistBins = i*RDbinSize;
        RadialDistHist = 0;
        Jacobian = (4*3.14159/3)*(RadialDistBins+RDbinSize)*(RadialDistBins+RDbinSize)*(
            ↪ RadialDistBins+RDbinSize) - (4*3.14159/3)*(RadialDistBins*RadialDistBins*
            ↪ RadialDistBins);
        string AtomName;
        if (doclust==1)
        createMolObjects(&Waters, "residue(WAT)", firstpass, watcount, "water");
        //Creating lists of pointers to objects;
        if (type==1) //res-res type.
        {
                if (firstpass==1)
                {
                        S1->apply(sel_obj1);
                        for (res_it = S1->beginResidue(); res_it != S1->endResidue(); ++res_it)
                        {
                                if(res_it->isSelected())
                                {
                                        CurrentHist->ResList1.push_back(&(*res_it));
                                }
                        }
                        S1->deselect();
                        S1->apply(sel_obj2);
                        for (res_it = S1->beginResidue(); res_it != S1->endResidue(); ++res_it)
                        {
                                if(res_it->isSelected())
                                CurrentHist->ResList2.push_back(&(*res_it));
                        }
                        S1->deselect();
                }
                cout<<"\n  CurrentHist->ResList1.size()= "<<CurrentHist->ResList1.size()<<endl;
                for (int liI=0;liI<int(CurrentHist->ResList1.size());liI++)
                {
                        CurrentHist->ResList1[liI]->apply(Cent_Res);
                        center_of_mass_i = Cent_Res.getCenter();
                        for (int liJ=0;liJ<int(CurrentHist->ResList2.size());liJ++)
                        {
                                if (CurrentHist->ResList1[liI]!=CurrentHist->ResList2[liJ])
                                {
                                        CurrentHist->ResList2[liJ]->apply(Cent_Res);
```

```
                                        center_of_mass_j = Cent_Res.getCenter();
                                        diff_ij = center_of_mass_j - center_of_mass_i;
                                        //Take into avount periodic boundary conditions
                                        MinimumImage(diff_ij,framenumber);
                                        ldDist = diff_ij.getLength();
                                        for (int liZ=0;liZ<(RDsize-1);liZ++)
                                        {
                                                if (ldDist<RadialDistBins(liZ+1))
                                                {
                                                        RadialDistHist(liZ)++;
                                                        break;
                                                }
                                                else continue;
                                        }
                                        if (doclust==1)
                                        {
                                                ss.clear();
                                                ss<<CurrentHist->ResList1[liI]->getID();
                                                ss>>ID1;
                                                ss.clear();
                                                ss<<CurrentHist->ResList2[liJ]->getID();
                                                ss>>ID2;
                                                FillWatLink(ldDist, ID1, ID2);
                                        }

                                }
                        }
                }
                resicount = int (CurrentHist->ResList1.size());
                resjcount = int (CurrentHist->ResList2.size());
        }
        else if (type==2) //atom-atom type.
        {
                if (firstpass==1)
                {
                        S1->apply(sel_obj1);
                        for (at_it = S1->beginAtom(); at_it != S1->endAtom(); ++at_it)
                        {
                                if(at_it->isSelected())
                                {
                                        AtomName = at_it->getName();
                                        //cout<<"  Found atom has name: "<<AtomName<<endl;
                                        CurrentHist->AtomList1.push_back(at_it->getIndex());
                                }
                        }
                        S1->deselect();
                        S1->apply(sel_obj2);
                        for (at_it = S1->beginAtom(); at_it != S1->endAtom(); ++at_it)
                        {
                                if(at_it->isSelected())
                                {
                                        AtomName = at_it->getName();
                                        CurrentHist->AtomList2.push_back(at_it->getIndex());
                                }
                        }
                        S1->deselect();
                        cout<<"  AtomList1.size() = "<<CurrentHist->AtomList1.size()<<", AtomList2.
                            ↪ size() = "<<CurrentHist->AtomList2.size()<<endl;
                }
                resicount = int (CurrentHist->AtomList1.size());
                resjcount = int (CurrentHist->AtomList2.size());
                if (resicount!=0 && resjcount!=0)
                {
                        speciesfound=1;
                        for (int liI=0;liI<int(CurrentHist->AtomList1.size());liI++)
                        {
                                Atom::StaticAtomAttributes *d1 = &(Atom::getAttributes()[CurrentHist
                                    ↪ ->AtomList1[liI]]);
                                res_i = d1->ptr->getResidue();
                                for (int liJ=0;liJ<int(CurrentHist->AtomList2.size());liJ++)
                                {
```

```
                                        Atom::StaticAtomAttributes *d2 =&(Atom::getAttributes()[
                                            ↪ CurrentHist->AtomList2[liJ]]);
                                        res_j = d2->ptr->getResidue();
                                        if ((res_i!=res_j) && (CurrentHist->AtomList1[liI]!=
                                            ↪ CurrentHist->AtomList2[liJ]) )
                                        {
                                                diff_ij = (d1->position) - (d2->position);
                                                //Take into avount periodic boundary conditions
                                                MinimumImage(diff_ij,framenumber);
                                                ldDist = diff_ij.getLength();
                                                for (int liZ=0;liZ<(RDsize-1);liZ++)
                                                {
                                                        if (ldDist<RadialDistBins(liZ+1))
                                                        {
                                                                RadialDistHist(liZ)++;
                                                                break;
                                                        }
                                                        else continue;
                                                }
                                                if (doclust==1) //If Clusterization is ON, we need
                                                    ↪ to create water neighbours lists
                                                {
                                                        ss.clear();
                                                        ss<<res_i->getID();

                                                        ss>>ID1;
                                                        ss.clear();
                                                        ss>>ID2;
                                                        FillWatLink(ldDist, ID1, ID2);
                                                }
                                        }//res_i!=res_j
                                } //liJ cycle
                        } //liI cycle
                }//resjcount!=0, resicount!=0
        }//type==2
        else
        {
                cout<<"  ERROR: undefined obj-obj relation in variable - int type.\nNothing can be
                    ↪ done, exit.\n";
                return 0;
        }
//Normalizing Radial distribution
        if (speciesfound==1)
        {
                ldDensity2 = resjcount/((MaxX-MinX)*(MaxY-MinY)*(MaxZ-MinZ));
                if(ldDensity2==0)
                {
                        cout<<"  ZERO density! ldDensity2="<< ldDensity2<<", resjcount = "<<
                            ↪ resjcount<<".ldDensity2 will be set to 1"<< endl;
                        ldDensity2=1;
                }
                //Normalizing of RDF.
                RadialDistHist=RadialDistHist/resicount;
                RadialDistHist=RadialDistHist/(ldDensity2*Jacobian);
        }
//Writing values to Averages.
        if (firstpass==1)
        {
                CurrentHist->selector2 = nameforselect2;
                CurrentHist->selector1 = nameforselect1;
                CurrentHist->outname = outname;
                CurrentHist->RadialDistBins.resize(RadialDistBins.shape());
                CurrentHist->RadialDistBins = RadialDistBins;
                CurrentHist->RadialDistHist.resize(RadialDistHist.shape());
                CurrentHist->RadialDistHist = RadialDistHist;
                if (speciesfound==1)
                {
                        CurrentHist->framnum = 1;
                }
                else
                {
                        CurrentHist->framnum = 0;
```

```
                        return 0;
                }
        }
        else
        {
                if (speciesfound==1)
                {
                CurrentHist->RadialDistHist = CurrentHist->RadialDistHist + RadialDistHist;
                CurrentHist->framnum++;
                }
                else return 0;
        }
//Writing averages to the file
        ss.clear();
        ss<<outname<<"RadialDist.txt";
        char outname2[200];
        ss>>outname2;
        cout<<"  Writing results to the file: "<<outname2<<endl;
        ofstream outfile1;
        outfile1.open(outname2);
        outfile1<<"# Radial Distribution Function for selector1 = "<<nameforselect1<<" and selector2
            ↪  = "<<nameforselect2<<".\n# First Column Bins (distance),\n# Second Column is N(R)
            ↪ /(4*pi*R^2*dens*binsize).\n";
        outfile1<<"# Density of '"<<nameforselect2<<"' =N2/V : "<<ldDensity2<<endl;
        for (int liZ=0;liZ<RDsize;liZ++)
        {
                //cout<<"\n "<<CurrentHist->RadialDistBins(liZ)<<" "<<CurrentHist->RadialDistHist(
                    ↪ liZ);
                if (isinf(CurrentHist->RadialDistHist(liZ)))
                cout<<"  INFINITE value of RadialDistHist("<<liZ<<")= "<<CurrentHist->RadialDistHist
                    ↪ (liZ)<<endl;
                if (isnan(CurrentHist->RadialDistHist(liZ)))
                CurrentHist->RadialDistHist(liZ) = 0;
                outfile1<<"\n "<<CurrentHist->RadialDistBins(liZ)<<" "<<CurrentHist->RadialDistHist(
                    ↪ liZ);
        }
        outfile1.clear();
        outfile1.close();
        S1->deselect();
        return 1;
}
void Snapshot::FillWatLink(double ldDist, int ID_i, int ID_j)  //While calculating create neighbor
    ↪ lists for each atom of interest
{...}

void Snapshot::FillSurfLink(double ldDist, int ID_i, int ID_j)//While calculating create neighbor
    ↪ lists for each atom of interest
{...}

void Snapshot::FillWatSurfLink(double ldDist, int ID_i, int ID_j)//While calculating create neighbor
    ↪  lists for each atom of interest
{...}

void Snapshot::Clustering (Averages* Ave, bool firstpass, int ClustCount)//Clustering of water-water
    ↪  1D
{
        cout<<"FUNCTION CALL: Clusterization\n";
        bool firstpass2=1;
        bool isin1;
        bool isin2;
        double ClDbinSize=1;
        //Initialization of Cluster Distribution function
        int MaxValue = 4000;
        int ClDsize = int(MaxValue/ClDbinSize)+1;
        Array<double,1> ClusterDistBins(ClDsize);
        Array<double,1> ClusterDistHist(ClDsize);
        firstIndex i;
        ClusterDistBins = i*ClDbinSize;
        ClusterDistHist = 0;
        Listall.clear();
        int ClustZerosize=0;
        char clustername[70];
```

```
        if (firstpass==1)
        { // Allocate memory for the first run of clustering
                Cluster = new list<int>[ClustCount];
        }
        int inClustCount=0;
        for (int liK=0;liK<WatersSize;liK++)
                {
                        Listall.push_back(Waters[liK].ID);
                }
                Listall.sort();
                Listall.unique();
        for (int liJ=0;liJ<ClustCount;liJ++)
        {
                sprintf(clustername,"./Cluster%d.txt",liJ);
                firstpass2=1;
                Cluster[liJ].clear();//clear for each new snapshot
//  cout<< "\nFirst sorting. ";
                for (int liI=0;liI<WatersSize;liI++)
                {
                        isin1 = IsInList(Listall, Waters[liI].ID);
                        if (isin1==1)
                        {
                                if (firstpass2==1)
                                {
                                        Cluster[liJ].push_back(Waters[liI].ID);
                                        Listall.remove(Waters[liI].ID);
                                        isin2=IsInList(Listall,Waters[liI].ID);
                                        for (ljt=Waters[liI].LinkswithWat.begin();ljt!=Waters[liI].
                                            ↪ LinkswithWat.end();ljt++)
                                        {
                                                Cluster[liJ].push_back(*ljt);
                                                Listall.remove(*ljt);
                                                isin2=IsInList(Listall,*ljt);//just checking whether
                                                    ↪  deleted.This line can be removed.
                                        }
                                        firstpass2=0;
                                }
                                else
                                {
                                        isin1 = IsInList(Cluster[liJ],Waters[liI].ID);
                                        if (isin1==1)
                                        {
                                                for (ljt=Waters[liI].LinkswithWat.begin();ljt!=
                                                    ↪ Waters[liI].LinkswithWat.end();ljt++)
                                                {
                                                        Cluster[liJ].push_back(*ljt);
                                                        Listall.remove(*ljt);
                                                        isin2=IsInList(Listall,*ljt);//just checking
                                                            ↪  whether deleted.This line can be
                                                            ↪ removed.
                                                }
                                        }
                                        else continue;
                                }
                        }
                        else
                        {
                                isin1 = IsInList(Cluster[liJ],Waters[liI].ID);
                                if (isin1==1)
                                {
                                        for (ljt=Waters[liI].LinkswithWat.begin();ljt!=Waters[liI].
                                            ↪ LinkswithWat.end();ljt++)
                                        {
                                                Cluster[liJ].push_back(*ljt);
                                                Listall.remove(*ljt);
                                                isin2=IsInList(Listall,*ljt);//just checking whether
                                                    ↪  deleted.This line can be removed.
                                        }
                                }
                        }
                }
                Cluster[liJ].sort();//sorting from low to large
```

```
                Cluster[liJ].unique();//leaving only unique values
                Listall.sort();
                Listall.unique();
//Sorting and adding neighbors of neighbors. It will be repeated until cluster size stops changing.
                int SizeBefore=0;
                int k=1;//counter of sorting/adding repetition. First adding/sorting has been done.
                while (SizeBefore!=(int)Cluster[liJ].size())
                {
                        k++;
                        SizeBefore=(int)Cluster[liJ].size();
                        //cout<< "\nSorting #"<<k<<" Including the neigbors of the neigbors.";
                        for (int liI=0;liI<WatersSize;liI++)
                        {
                                isin1 = IsInList(Listall, Waters[liI].ID);
                                if (isin1==1)
                                {
                                        isin1 = IsInList(Cluster[liJ],Waters[liI].ID);
                                        if (isin1==1)
                                        {
                                                for (ljt=Waters[liI].LinkswithWat.begin();ljt!=
                                                    ↪ Waters[liI].LinkswithWat.end();ljt++)
                                                {
                                                        Cluster[liJ].push_back(*ljt);
                                                        Listall.remove(*ljt);
                                                        isin2=IsInList(Listall,*ljt);//just checking
                                                            ↪  whether deleted.This line can be
                                                            ↪ removed.
                                                }
                                        }
                                        else continue;
                                }
                                else
                                {
                                        isin1 = IsInList(Cluster[liJ],Waters[liI].ID);
                                        if (isin1==1)
                                        {
                                                for (ljt=Waters[liI].LinkswithWat.begin();ljt!=
                                                    ↪ Waters[liI].LinkswithWat.end();ljt++)
                                                {
                                                        Cluster[liJ].push_back(*ljt);
                                                        Listall.remove(*ljt);
                                                        isin2=IsInList(Listall,*ljt);//just checking
                                                            ↪  whether deleted.This line can be
                                                            ↪ removed.
                                                }
                                        }
                                }
                        }
                        Cluster[liJ].sort();//sorting from low to large
                        Cluster[liJ].unique();//leaving only unique values
                        Listall.sort();
                        Listall.unique();
                }
//Cluster histogramming. We also need to divide it by number of NON-ZERO clusters to normalize
    ↪ distribution.
                for (int liZ=0; liZ<ClDsize;liZ++)
                {
                        if ((int(Cluster[liJ].size())<ClusterDistBins(liZ)||int(Cluster[liJ].size())
                            ↪ ==ClusterDistBins(liZ))&&Cluster[liJ].size()!=0)
                        {
                                ClusterDistHist(liZ)++;
                                break;
                        }
                        else continue;
                }
                inClustCount=inClustCount+(int)Cluster[liJ].size(); //NOTE: it will be calculated
                    ↪ n_tot times.
//Calculating number of zero size clusters to eliminate them during normalization
                if (Cluster[liJ].size()==0)
                ClustZerosize++;
                else
                {
```

```
                        cout<<"  Cluster # "<<liJ<<" : "<<Cluster[liJ].size()<<" waters.\n";
                        cout<<"    Wat_in = { ";
                        for (lit=Cluster[liJ].begin();lit!=Cluster[liJ].end();lit++)
                        {
                                cout<<*lit<<" ";
                        }
                        cout<<" }\n";
                }
        }
        cout<<"\n  Cluster objects are created. Number of zero size clust = "<<ClustZerosize<<endl;
        //Normalizing cluster distibution and writing it to the file.
        ClusterDistHist=ClusterDistHist/(ClustCount-ClustZerosize);
    myoutfile.open("./ClusterDistribution.txt");
        myoutfile<<"# Cluster Size Distribution Function.\n# First Column Bines (number of molecules
            ↪  in cluster,\n# Second Column is normalized number of clusters with this number of
            ↪ molecules in them.";
        for (int liI=0;liI<ClDsize;liI++)
        {
                myoutfile<<"\n "<<ClusterDistBins(liI)<<" "<<ClusterDistHist(liI);
        }
        myoutfile.clear();
        myoutfile.close();
        //Summary of clusterization.
        cout<<"  Overall number of wat_mol inluded to "<<(ClustCount-ClustZerosize)<<" clusters is "
            ↪ <<inClustCount;
        cout<<". Number of wat_mol left is "<<(int)Listall.size()<<endl;
        //Writing values to Averages.
        if (firstpass==1)
        {
                Ave->ClusterDistBins.resize(ClusterDistBins.shape());
                Ave->ClusterDistBins = ClusterDistBins;
                Ave->ClusterDistHist.resize(ClusterDistHist.shape());
                Ave->ClusterDistHist = ClusterDistHist;
        }
        else
        {
                Ave->ClusterDistHist = Ave->ClusterDistHist + ClusterDistHist;

        }
}

vector<double> Snapshot::DefineSlab(int liWhere) // Find and approximate spatial domains - where we
    ↪ have mostly oil, surfactant or water layers.
{...}

bool Snapshot::WS_clustering2D(Averages* Ave, Hist* CurrentHist, bool firstpass, int ClustCount)//
    ↪ Clustering taking into account both water and surfactant hydrogen bonds
{
        string nameforselect1 = "residue(WAT) AND name(O)";
        Selector sel_obj1(nameforselect1);
        string nameforselect2 = "residue(ant) AND (name(C1) OR name(C2) OR name(C3) OR name(C4) OR
            ↪ name(C5) OR name(C6) OR name(C9) OR name(C10) OR name(C11) OR name(C15) OR name(O12)
            ↪ OR name(C16) OR name(O17) OR name(C18) OR name(C19) OR name(O20) OR name(C21) OR name
            ↪ (C22) OR name(O23))";
        Selector sel_obj2(nameforselect2);
        double ldDist;
        int icount=0;
        int jcount=0;
        Vector3 center_of_mass_i(0,0,0);
        Vector3 center_of_mass_j(0,0,0);
        Vector3 diff_ij(0,0,0);
        Element AtomElem1;
        Residue *res_i, *res_j;
        Atom::StaticAtomAttributes *d1,*d2;
        string objname1,objname2;
        stringstream ss (stringstream::in | stringstream::out);
        vector<double> SlabZ;
        int ID1=0, ID2=0;

        createMolObjects(&Surfactants, "residue(ant)", firstpass, surfcount, "surfactant");
        createMolObjects(&Waters, "residue(WAT)", firstpass, watcount, "water");

```

```
        //to find vertical coordinates where we're going to calculate RD
        cout<<"FUNCTION CALL: WS_clustering2D with maximum number of clusters: "<<ClustCount<<endl;
        bool isin1;
        bool isin2;
        double ClDbinSize=1;
        //Initialization of Cluster Distribution function
        int MaxValue = 40;
        int ClDsize = int(MaxValue/ClDbinSize)+1;
        Array<double,1> ClusterDistBins(ClDsize);
        Array<double,1> ClusterDistHist_Wat(ClDsize);
        Array<double,1>* ClusterDistHist_Surf;
        ClusterDistHist_Surf = new Array<double,1>[ClDsize];
        for (int liI=0;liI<ClDsize; liI++)
        {
                ClusterDistHist_Surf[liI].resize(ClDsize);
                ClusterDistHist_Surf[liI] = 0;
        }
        firstIndex i;
        ClusterDistBins = i*ClDbinSize;
        ClusterDistHist_Wat = 0;
        Listall.clear();
        Listall_surf.clear();
        int ClustZerosize=0;
        char clustername[70];

        if (firstpass==1)
        {
                WatSurfCl = new Cluster2D[ClustCount];
                WS_Cl_index = 0;

        }
        int inClustCount=0;
        SW_Cl_index = 0;
        SlabZ = DefineSlab(1);//1 - in oil-surfactant slab.
        cout<<" Oil slab is located z= {";
        for (int liI=0; liI<int(SlabZ.size());liI++)
        {
                cout<<" "<<SlabZ[liI];
        }
        cout<<" }.\n}";
        if (int(SlabZ.size())%2!=0) //vector is not contracted of pairs z_begin and z_end.
        {
                cout<<"  ERROR: SlabZ has wrong size: "<<SlabZ.size();
                return 0;
        }
        for (int liJ=0;liJ< int(SlabZ.size());liJ+=2)
        {
                if(int(SlabZ.size())!=2)
                cout<<"  Histogramming # "<<liJ+1<<" slab: searching for heavy water and surfactant
                    ↪ atoms in "<<SlabZ[liJ]<<" < z < "<<SlabZ[liJ+1]<<endl;

                CurrentHist->AtomList1.clear();
                CurrentHist->AtomList2.clear();
                S1->apply(sel_obj1);
                for (at_it = S1->beginAtom(); at_it != S1->endAtom(); ++at_it)
                {
                        if(at_it->isSelected())
                        {
                                center_of_mass_i = at_it->getPosition();
                                if (center_of_mass_i.z>SlabZ[liJ]&&center_of_mass_i.z<SlabZ[liJ+1])
                                CurrentHist->AtomList1.push_back(at_it->getIndex());
                        }
                }
                icount = int (CurrentHist->AtomList1.size());
                cout<<"  AtomList1.size() = "<<icount<<endl;
                S1->deselect();
                S1->apply(sel_obj2);
                for (at_it = S1->beginAtom(); at_it != S1->endAtom(); ++at_it)
                {
                        if(at_it->isSelected())
                        {
                                center_of_mass_i = at_it->getPosition();
```

```
                        if (center_of_mass_i.z>SlabZ[liJ]&&center_of_mass_i.z<SlabZ[liJ+1])
                        CurrentHist->AtomList2.push_back(at_it->getIndex());
                }
        }
        jcount = int (CurrentHist->AtomList2.size());
        cout<<"  AtomList2.size() = "<<jcount<<endl;
        S1->deselect();
        if (icount==0)
        {
                cout<<"  No water found in the oil slab. Skip to next frame.\n";
                return 0;
        }
        else if (jcount==0)
        {
                cout<<"  No sufractant found in the oil slab. Skip to next frame.\n";
                return 0;
        }
        else
        { //else - molecules found in the oil slab
                for (int liI=0;liI<int(CurrentHist->AtomList1.size());liI++)
                {//i loop
                        Atom::StaticAtomAttributes *d1 = &(Atom::getAttributes()[CurrentHist
                            ↪ ->AtomList1[liI]]);
                        res_i = d1->ptr->getResidue();
                        //Searching for links between water molecules
                        for (int liJ=0;liJ<int(CurrentHist->AtomList1.size());liJ++)
                        {
                                Atom::StaticAtomAttributes *d2 =&(Atom::getAttributes()[
                                    ↪ CurrentHist->AtomList1[liJ]]);
                                res_j = d2->ptr->getResidue();
                                if ((res_i!=res_j) && (CurrentHist->AtomList1[liI]!=
                                    ↪ CurrentHist->AtomList1[liJ]) )
                                {
                                        diff_ij = (d1->position) - (d2->position);
                                        ldDist = diff_ij.getLength();
                                        ss.clear();
                                        ss<<res_i->getID();
                                        ss>>ID1;
                                        ss.clear();
                                        ss<<res_j->getID();
                                        ss>>ID2;
                                        FillWatLink(ldDist, ID1, ID2);
                                }
                        } //first j loop (in wat list)
                        //Searching for links between water and surfactant molecules
                        for (int liJ=0;liJ<int(CurrentHist->AtomList2.size());liJ++)
                        {
                                Atom::StaticAtomAttributes *d2 =&(Atom::getAttributes()[
                                    ↪ CurrentHist->AtomList2[liJ]]);
                                res_j = d2->ptr->getResidue();
                                if ((res_i!=res_j) && (CurrentHist->AtomList1[liI]!=
                                    ↪ CurrentHist->AtomList2[liJ]) )
                                {
                                        diff_ij = (d1->position) - (d2->position);
                                        ldDist = diff_ij.getLength();
                                        ss.clear();
                                        ss<<res_i->getID();
                                        ss>>ID1;
                                        ss.clear();
                                        ss<<res_j->getID();
                                        ss>>ID2;
                                        FillWatSurfLink(ldDist, ID1, ID2);//also fills the
                                            ↪ surfactant list.
                                }
                        }       //second j loop (in surf list)
                }// i loop
                for (int liI=0;liI<int(CurrentHist->AtomList2.size());liI++)
                {//i loop
                        Atom::StaticAtomAttributes *d1 = &(Atom::getAttributes()[CurrentHist
                            ↪ ->AtomList2[liI]]);
                        res_i = d1->ptr->getResidue();
                        for (int liJ=0;liJ<int(CurrentHist->AtomList2.size());liJ++)
```

```
                                {
                                        Atom::StaticAtomAttributes *d2 =&(Atom::getAttributes()[
                                            ↪ CurrentHist->AtomList2[liJ]]);
                                        res_j = d2->ptr->getResidue();
                                        if ((res_i!=res_j) && (CurrentHist->AtomList2[liI]!=
                                            ↪ CurrentHist->AtomList2[liJ]) )
                                        {
                                                diff_ij = (d1->position) - (d2->position);
                                                ldDist = diff_ij.getLength();
                                                ss.clear();
                                                ss<<res_i->getID();
                                                ss>>ID1;
                                                ss.clear();
                                                ss<<res_j->getID();
                                                ss>>ID2;
                                                FillSurfLink(ldDist, ID1, ID2);//also fills the
                                                    ↪ surfactant list.
                                        }
                                }
                        }
                }//else - molecules found in the oil slab
        } //slab cycle.
        Listall.unique();
        Listall_surf.sort();
        Listall_surf.unique();
        cout<<"  There are "<<Listall.size()<<" waters and "<<Listall_surf.size()<<" surf-s involved
            ↪  in clustering; \n";
        WS_Cl_index = double(Listall.size())/double(Listall_surf.size()); //if you don't convert
            ↪ size type to float the answer will be zero.
        //cout<<"  WS_Cl_index = Number of water found/number of srf found = "<<WS_Cl_index<<endl;
        ParticleandLinks* WS;
        WS = new ParticleandLinks[watcount+surfcount];
        for (int liI=0;liI<watcount;liI++)
        {
                WS[liI]= Waters[liI];
        }
        for (int liI=watcount;liI<(watcount+surfcount);liI++)
        {
                WS[liI]= Surfactants[liI-watcount];
        }
        bool firstsorting;

        for (int liJ=0;liJ<ClustCount;liJ++)
        {
                //Sorting and adding neighbors of neighbors. It will be repeated liSortNum times.
                int WatSizeBefore=-1;
                int SurfSizeBefore=-1;
                int k=0;//counter of sorting/adding repetition. .
                sprintf(clustername,"./Cluster%d.txt",liJ);
                cout<<"\n*************** Cluster # "<<liJ<<" *************************\n";
                firstsorting = 1;
                //clear lists for each new snapshot
                WatSurfCl[liJ].Water_in.clear();
                WatSurfCl[liJ].Surf_in.clear();
                while (WatSizeBefore!=(int)WatSurfCl[liJ].Water_in.size()||SurfSizeBefore!=WatSurfCl
                    ↪ [liJ].Surf_in.size())
                {
                        k++;
                        WatSizeBefore = (int)WatSurfCl[liJ].Water_in.size();
                        SurfSizeBefore = int(WatSurfCl[liJ].Surf_in.size());
                        //cout<< "\nSorting #"<<k<<" Including the neigbors of the neigbors.";
                        for (int liI=0;liI<(watcount+surfcount);liI++)
                        {
                                if (WS[liI].type == "water")
                                {
                                        isin1 = IsInList(Listall, WS[liI].ID);
                                }
                                else if (WS[liI].type == "surfactant")
                                {
                                        isin1 = IsInList(Listall_surf, WS[liI].ID);
                                }
                                if (isin1==1)
```

```
                {
                        if (firstsorting==1)
                        {
                                if (WS[liI].type == "water")
                                {
                                        WatSurfCl[liJ].Water_in.push_back(WS[liI].ID
                                            ↪ );
                                        Listall.remove(WS[liI].ID);
                                }
                                else if (WS[liI].type == "surfactant")
                                {
                                        WatSurfCl[liJ].Surf_in.push_back(WS[liI].ID)
                                            ↪ ;
                                        Listall_surf.remove(WS[liI].ID);
                                }
                                for (ljt=WS[liI].LinkswithWat.begin();ljt!=WS[liI].
                                    ↪ LinkswithWat.end();ljt++)
                                {
                                        WatSurfCl[liJ].Water_in.push_back(*ljt);
                                        Listall.remove(*ljt);
                                        //isin2=IsInList(Listall,*ljt);//just
                                            ↪ checking whether deleted.This line
                                            ↪ can be removed.
                                }
                                for (ljt=WS[liI].LinkswithSurf.begin();ljt!=WS[liI].
                                    ↪ LinkswithSurf.end();ljt++)
                                {
                                        WatSurfCl[liJ].Surf_in.push_back(*ljt);
                                        Listall_surf.remove(*ljt);
                                        //isin2=IsInList(Listall_surf,*ljt);//just
                                            ↪ checking whether deleted.This line
                                            ↪ can be removed.
                                }
                                firstsorting=0;
                        }
                        else
                        {
                                if (WS[liI].type == "water")
                                {
                                        isin1 = IsInList(WatSurfCl[liJ].Water_in, WS
                                            ↪ [liI].ID);
                                }
                                else if (WS[liI].type == "surfactant")
                                {
                                        isin1 = IsInList(WatSurfCl[liJ].Surf_in, WS[
                                            ↪ liI].ID);
                                }
                                if (isin1==1)
                                {
                                        for (ljt=WS[liI].LinkswithWat.begin();ljt!=
                                            ↪ WS[liI].LinkswithWat.end();ljt++)
                                        {
                                                WatSurfCl[liJ].Water_in.push_back(*
                                                    ↪ ljt);
                                                Listall.remove(*ljt);
                                                //isin2=IsInList(Listall,*ljt);//
                                                    ↪ just checking whether deleted
                                                    ↪ .This line can be removed.
                                        }
                                        for (ljt=WS[liI].LinkswithSurf.begin();ljt!=
                                            ↪ WS[liI].LinkswithSurf.end();ljt++)
                                        {
                                                WatSurfCl[liJ].Surf_in.push_back(*
                                                    ↪ ljt);
                                                Listall_surf.remove(*ljt);
                                                //isin2=IsInList(Listall_surf,*ljt)
                                                    ↪ ;//just checking whether
                                                    ↪ deleted.This line can be
                                                    ↪ removed.
                                        }
                                }
                                else continue;
```

```
                                }
                        }
                        else
                        {
                                if (WS[liI].type == "water")
                                {
                                        isin1 = IsInList(WatSurfCl[liJ].Water_in, WS[liI].ID
                                            ↪ );
                                }
                                else if (WS[liI].type == "surfactant")
                                {
                                        isin1 = IsInList(WatSurfCl[liJ].Surf_in, WS[liI].ID)
                                            ↪ ;
                                }
                                if (isin1==1)
                                {
                                        for (ljt=WS[liI].LinkswithWat.begin();ljt!=WS[liI].
                                            ↪ LinkswithWat.end();ljt++)
                                        {
                                                WatSurfCl[liJ].Water_in.push_back(*ljt);
                                                Listall.remove(*ljt);
                                                //isin2=IsInList(Listall,*ljt);//just
                                                    ↪ checking whether deleted.This line
                                                    ↪ can be removed.
                                        }
                                        for (ljt=WS[liI].LinkswithSurf.begin();ljt!=WS[liI].
                                            ↪ LinkswithSurf.end();ljt++)
                                        {
                                                WatSurfCl[liJ].Surf_in.push_back(*ljt);
                                                Listall_surf.remove(*ljt);
                                                //isin2=IsInList(Listall_surf,*ljt);//just
                                                    ↪ checking whether deleted.This line
                                                    ↪ can be removed.
                                        }
                                }
                                else continue;

                        }
                }
                WatSurfCl[liJ].Surf_in.sort();//sorting from low to large
                WatSurfCl[liJ].Surf_in.unique();//leaving only unique values
                WatSurfCl[liJ].Water_in.sort();
                WatSurfCl[liJ].Water_in.unique();
                Listall.sort();
                Listall.unique();
                Listall_surf.sort();
                Listall_surf.unique();

        for (int liZ=0; liZ<ClDsize;liZ++)
        {
                //myoutfile<<"\n["<<liZ<<"] "<<ClusterDistBins[liZ]<<" "<<ClusterDistHist[
                    ↪ liZ];

                if (int(WatSurfCl[liJ].Water_in.size())<=ClusterDistBins(liZ)&&WatSurfCl[liJ
                    ↪ ].Water_in.size()!=0)
                {
                        //cout<<"# "<<liJ<<" Cluster has "<<Watsize<<" waters and "<<
                            ↪ Surfsize<<" surfactants\n";
                        ClusterDistHist_Wat(liZ)++;
                        // cout<<"\n cluster #"<<liJ<<" histogram ++";
                        for (int liX=0; liX<ClDsize;liX++) //there are P_surf distribution
                            ↪ for each water cluster size.
                        {
                                if (int(WatSurfCl[liJ].Surf_in.size())<=ClusterDistBins(liX)
                                    ↪ )
                                {
                                        ClusterDistHist_Surf[liZ](liX)++;
                                        break;
                                }
                                else continue;
                        }
                        break;
```

```
800                         }
                            else continue;
                    }
                    inClustCount=inClustCount+(int)WatSurfCl[liJ].Water_in.size(); //NOTE: it will be
                        ↪ calculated n_tot times.
                    int Watsize = WatSurfCl[liJ].Water_in.size();
                    int Surfsize = WatSurfCl[liJ].Surf_in.size();
//Calculating number of zero size clusters to eliminate them during normalization
                    if (WatSurfCl[liJ].Water_in.size()==0)
                    {
                            ClustZerosize++;
                            if (Surfsize!=0)
                            cout<<"  Cluster # "<<liJ<<" : "<<Watsize<<" waters, "<<Surfsize<<"
                                ↪ surfactants.\n";

                    }
                    else
                    {
                            SW_Cl_index+=double(Surfsize)/double(Watsize);
                            cout<<"  Cluster # "<<liJ<<" : "<<Watsize<<" waters, "<<Surfsize<<"
                                ↪ surfactants.\n";
                            cout<<"    Wat_in = { ";
                            for (lit=WatSurfCl[liJ].Water_in.begin();lit!=WatSurfCl[liJ].Water_in.end();
                                ↪ lit++)
                            {
                                    cout<<*lit<<" ";
                            }
                            cout<<" }\n";
                    }
            }//cluster [liJ] loop
            cout<<"  Cluster objects are created. Number of zero size clust = "<<ClustZerosize<<endl;
//Normalizing cluster distibution and writing it to the file.
            for (int liI=0;liI<ClDsize;liI++)
            { //ClusterDistHist_Wat(liI) gives us how many clusters have liI water molecules. This is
                ↪ the m =normalization factor for the Pm(n) distribution that says what is the
                ↪ probability to have n srf molecules if there are m water molecules in the cluster.
                    if (ClusterDistHist_Wat(liI)!=0)
                    ClusterDistHist_Surf[liI]=ClusterDistHist_Surf[liI]/(ClusterDistHist_Wat(liI));
                    char name[50];
                    ss.clear();
                    ss<<ClusterDistBins(liI)<<"SurfClustDist.txt";
                    ss>>name;
                    myoutfile.clear();
                    myoutfile.open(name);
                    myoutfile<<"# Cluster Size Distribution Function.\n# This gives the probability to a
                        ↪  cluster to have certain number of surfactants if the water number is "<<
                        ↪ ClusterDistBins(liI)<<". \n";
                    myoutfile<<"#Number of clusters with"<<liI<<" waters in this frame is"<<
                        ↪ ClusterDistHist_Wat(liI)<<".  \n";
                    myoutfile<<"#1st Column: number of molecules in cluster; 2nd Column: normalized
                        ↪ probarility P_n_surf(given m_water)";
                    for (int liJ=0; liJ<ClDsize;liJ++)
                    myoutfile<<"\n "<<ClusterDistBins(liJ)<<" "<< ClusterDistHist_Surf[liI](liJ);
            }
            myoutfile.close();
            myoutfile.clear();
//Writing usual P_m_water to the file;
            ClusterDistHist_Wat=ClusterDistHist_Wat/(ClustCount-ClustZerosize);
            SW_Cl_index = SW_Cl_index/(ClustCount-ClustZerosize);
            cout<<"  Number of surfactants per water molecule SW_Cl_index = "<<SW_Cl_index<<endl;
            myoutfile.open("WatClustDist.txt");
            myoutfile<<"# Cluster Size Distribution Function.\n# 1st Column: Bines (number of molecules
                ↪ in cluster;\n# 2nd Column: number of clusters which have so many water mol-s.";
            myoutfile<<"# Number of surfactants per one water molecule SW_Cl_index = "<<SW_Cl_index<<
                ↪ endl;
            for (int liI=0;liI<ClDsize;liI++)
            {
                    myoutfile<<"\n "<<ClusterDistBins(liI)<<" "<<ClusterDistHist_Wat(liI);
            }
            myoutfile.close();
            myoutfile.clear();
//Summary of clusterization.
```

```
        cout<<"  Overall number of wat_mol inluded to "<<(ClustCount-ClustZerosize)<<" clusters is "
            ↪ <<inClustCount<<".";
        cout<<"  Number of waters per surfactant: WS_ Cl_index = "<<WS_Cl_index<<endl;
//Writing values to Averages.
        if (firstpass==1)
        {
                Ave->ClusterDistHist_Surf = new Array<double,1>[ClDsize];
                for (int liI=0;liI<ClDsize; liI++)
                {
                        Ave->ClusterDistHist_Surf[liI].resize(ClusterDistHist_Surf[liI].shape());
                        Ave->ClusterDistHist_Surf[liI] = ClusterDistHist_Surf[liI];
                }
                Ave->ClusterDistBins.resize(ClusterDistBins.shape());
                Ave->ClusterDistBins = ClusterDistBins;
                Ave->ClusterDistHist_Wat.resize(ClusterDistHist_Wat.shape());
                Ave->ClusterDistHist_Wat = ClusterDistHist_Wat;
                Ave->WS_Cl_index = WS_Cl_index;
                Ave->SW_Cl_index = SW_Cl_index;
        }
        else
        {
                Ave->WS_Cl_index+= WS_Cl_index;
                Ave->SW_Cl_index+= SW_Cl_index;
                Ave->ClusterDistHist_Wat = Ave->ClusterDistHist_Wat + ClusterDistHist_Wat;
                for (int liI=0;liI<ClDsize; liI++)
                {
                        Ave->ClusterDistHist_Surf[liI] = Ave->ClusterDistHist_Surf[liI] +
                            ↪ ClusterDistHist_Surf[liI];
                }
        }
        delete [] WS, ClusterDistHist_Surf;
        return 1;
}

void Snapshot::SetBoxSize(string box_file_name, bool NVT) //In case our ensemble is NPT, assess
    ↪ average volume of the simulation box
{ //SetBox Size:
        char outname[200];
        stringstream ss (stringstream::in | stringstream::out);
        ss<<box_file_name;
        ss>>outname;
        myoutfile.open(outname);

        if (NVT==1)
        {  //NVT - constant volume trajectory.
                myoutfile>>MaxX>>MaxY>>MaxZ;
                Lbox = MaxZ;
                ss.clear();
                myoutfile.clear();
                myoutfile.close();
                cout<<"NVT Box size (Xmax,Ymax,Zmax) = ("<<MaxX<<", "<<MaxY<<", "<<MaxZ<<")\n";//.
                    ↪ Lbox = "<<Lbox<<".\n";
                Vconst=1; //i microem. simulations volume is constant.
        }
        else
        { //NPT - constant pressure trajectory.
                Vconst=0;
                double x=0;
                double y=0;
                double z=0;
                MaxX=0;
                MaxY=0;
                MaxZ=0;
                while(myoutfile)
                {
                        myoutfile>>x>>y>>z;
                        LboxX.push_back(x);
                        LboxY.push_back(y);
                        LboxZ.push_back(z);
                }
                for (int liI=0;liI<(int)LboxX.size();liI++)
                {
```

```
                        MaxX+=LboxX[liI];
                        MaxY+=LboxY[liI];
                        MaxZ+=LboxZ[liI];
                }
                MaxX=MaxX/int(LboxX.size());
                MaxY=MaxY/int(LboxY.size());
                MaxZ=MaxZ/int(LboxZ.size());
                Lbox=MaxZ;
                cout<<"NPT Average box size (Xmax,Ymax,Zmax) = ("<<MaxX<<", "<<MaxY<<", "<<MaxZ<<")\
                    ↪ n";
        }
}

void Snapshot::MinimumImage(Vector3& diff_ij, int framenumber) //Find the closest distance a given
    ↪ enclosed geometry  ( for Radial distribution functions)
{
        //searches for the nearest image of a particle given the periodic boundary conditions.
        if (Vconst==1)
        {
                if (fabs(diff_ij.x)>MaxX/2)
                {
                        if (diff_ij.x > 0)
                        {//Xi is larger than Xj. Transfer Xj for one box langth to the RIGHT -> -
                            ↪ MaxX in the difference.
                                diff_ij.x = diff_ij.x - MaxX;
                        }
                        else //Xi is smaller than Xj. Transfer Xj for one box langth to the LEFT ->
                            ↪ -MaxX in the difference.
                        diff_ij.x = diff_ij.x + MaxX;
                }
                if (fabs(diff_ij.y)>MaxY/2)
                {
                        if (diff_ij.y > 0)
                        {//Yi is larger than Yj. Transfer Yj for one box langth to the RIGHT -> -
                            ↪ MaxX in the difference.
                                diff_ij.y = diff_ij.y - MaxY;
                        }
                        else //Yi is smaller than Yj. Transfer Yj for one box langth to the LEFT ->
                            ↪ -MaxX in the difference.
                        diff_ij.y = diff_ij.y + MaxY;
                }
                if (fabs(diff_ij.z)>MaxZ/2)
                {
                        if (diff_ij.z > 0)
                        {//Zi is larger than Zj. Transfer Zj for one box langth to the RIGHT -> -
                            ↪ MaxX in the difference.
                                diff_ij.z = diff_ij.z - MaxZ;
                        }
                        else //Zi is smaller than Zj. Transfer Zj for one box langth to the LEFT ->
                            ↪ -MaxX in the difference.
                        diff_ij.z = diff_ij.z + MaxZ;
                }
        }
        else
        {
                if (fabs(diff_ij.x)>LboxX[framenumber]/2)
                {
                        if (diff_ij.x > 0)
                        {//Xi is larger than Xj. Transfer Xj for one box langth to the RIGHT -> -
                            ↪ MaxX in the difference.
                                diff_ij.x = diff_ij.x - LboxX[framenumber];
                        }
                        else //Xi is smaller than Xj. Transfer Xj for one box langth to the LEFT ->
                            ↪ -MaxX in the difference.
                        diff_ij.x = diff_ij.x + LboxX[framenumber];
                }
                if (fabs(diff_ij.y)>LboxY[framenumber]/2)
                {
                        if (diff_ij.y > 0)
                        {//Yi is larger than Yj. Transfer Yj for one box langth to the RIGHT -> -
                            ↪ MaxX in the difference.
                                diff_ij.y = diff_ij.y - LboxY[framenumber];
```

```
                        }
                        else //Yi is smaller than Yj. Transfer Yj for one box langth to the LEFT -> -MaxX in the difference.
                        diff_ij.y = diff_ij.y + LboxY[framenumber];
                }
                if (fabs(diff_ij.z)>LboxZ[framenumber]/2)
                {
                        if (diff_ij.z > 0)
                        {//Zi is larger than Zj. Transfer Zj for one box langth to the RIGHT -> -MaxX in the difference.
                                diff_ij.z = diff_ij.z - LboxZ[framenumber];
                        }
                        else //Zi is smaller than Zj. Transfer Zj for one box langth to the LEFT -> -MaxX in the difference.
                        diff_ij.z = diff_ij.z + LboxZ[framenumber];
                }
        }

}
```

3 Averages.cpp

```
#include "averages.h"
Averages::Averages()
{}
Averages::~Averages()
{}

void Averages::NormAndWrite(int HistCount, double LinkParam_ww, double LinkParam_ws)
{
        ofstream myoutfile;
        stringstream ss (stringstream::in | stringstream::out);
        char outname[200];
        int RDsize;
        cout<<"FUNC CALL::NormAndWrite \n";
//Normalizing and Writing RD's
        for (int liI=0;liI<HistCount;liI++)
        {
                allRDs[liI].RadialDistHist = allRDs[liI].RadialDistHist/allRDs[liI].framnum;
                RDsize = allRDs[liI].RadialDistBins.size();
                ss.clear();
                ss<<"Ave"<<allRDs[liI].outname<<"RadialDist.txt";
                ss>>outname;

                cout<<" Writing FINAL results to the file: "<<outname<<endl;
                myoutfile.open(outname);
                myoutfile<<"# Radial Distribution Function for selectors:"<<allRDs[liI].selector1<<" &" <<allRDs[liI].selector2<<" \n# Number of frames analyzed = "<<allRDs[liI].framnum;
                myoutfile<<".\n# First Column Bins (distance),\n# Second Column is g(r).\n";
                for (int liZ=0;liZ<RDsize;liZ++)
                {
                        if (isinf(allRDs[liI].RadialDistHist(liZ)))
                        cout<<"  INFINITE value of RadialDistHist("<<liZ<<")= "<<allRDs[liI].RadialDistHist(liZ)<<endl;
                        if (isnan(allRDs[liI].RadialDistHist(liZ)))
                        allRDs[liI].RadialDistHist(liZ) = 0;
                        myoutfile<<"\n "<<(allRDs[liI].RadialDistBins(liZ)+0.25)<<" "<<allRDs[liI].RadialDistHist(liZ);
                }
                ss.clear();
                myoutfile.clear();
                myoutfile.close();

        }
//Normalizing and Writing Cluster Distribution.
        myoutfile.clear();
        ClusterDistHist=ClusterDistHist/clustframenum;

        myoutfile.open("AVEClusterDistribution.txt");
```

```
        myoutfile<<"# Cluster Size Distribution Function.LinkParam_ww ="<<LinkParam_ww<<",
            ↪ LinkParam_ws= "<<LinkParam_ws<<" [angst]\n# First Column Bines (number of molecules
            ↪ in cluster,\n# Second Column is normalized number of clusters with this number of
            ↪ molecules in them.";
        int ClDsize = ClusterDistBins.size();

        for (int liI=0;liI<ClDsize;liI++)
        {
                myoutfile<<"\n "<<ClusterDistBins(liI)<<" "<<ClusterDistHist(liI);
        }
        myoutfile.clear();
        myoutfile.close();

//Normalizing and Writing 2D Cluster Distribution: WATER P(n)
        myoutfile.open("./Ave2DClustDist_Wat.txt");
        myoutfile<<"# Cluster Size Distribution Function.LinkParam ="<<LinkParam_ww<<"[angst]\n#
            ↪ First Column Bines (number of molecules in cluster,\n# Second Column is normalized
            ↪ number of clusters with this number of molecules in them.";
        myoutfile<<"\n# Number of frames analyzed by now: "<<clustframenum;
        myoutfile<<"# Number of water mol-s per one surfactant molecule WS_Cl_index = "<<WS_Cl_index
            ↪ /clustframenum<<endl;
        myoutfile<<"# Number of surfactant mol-s per one water molecues SW_Cl_index = "<<SW_Cl_index
            ↪ /clustframenum<<endl;
        int ClDsize = ClusterDistBins.size();

        for (int liI=0;liI<ClDsize;liI++)
        {
                myoutfile<<"\n "<<ClusterDistBins(liI)<<" "<<ClusterDistHist_Wat(liI)/clustframenum;
        }
        myoutfile.clear();
        myoutfile.close();

//Normalizing and Writing 2D Cluster Distribution: SURFACTANT P(m) for each n-water molecules in a
    ↪ cluster.
    for (int liI=0;liI<10;liI++)
        {
                char name[50];
                ss.clear();
                ss<<"./Ave"<<ClusterDistBins(liI)<<"SurfClustDist.txt";
                ss>>name;
                myoutfile.open(name);
                myoutfile<<"# Cluster Size Distribution Function. ";
                myoutfile<<"\n# LinkParam_ws ="<<LinkParam_ws<<"[angst], LinkParam_ww = "<<
                    ↪ LinkParam_ww<<endl;
                myoutfile<<"# This gives the probability to a cluster to have certain number of
                    ↪ surfactants if the water number is "<<ClusterDistBins(liI)<<".\n# 1st Column:
                    ↪  number of molecules in cluster; 2nd Column: normalized probarility P_n_surf(
                    ↪ given m_water)";
                myoutfile<<"\n# Number of frames analyzed by now: "<<clustframenum;
                for (int liJ=0;liJ<ClDsize;liJ++)
                {
                        myoutfile<<"\n "<<ClusterDistBins(liJ)<<" "<<ClusterDistHist_Surf[liI](liJ)/
                            ↪ clustframenum;
                }
                myoutfile.clear();
                myoutfile.close();
        }

//Normalizing and Writing Partition coefficicients.
        Ksurf = Ksurf/framecount;
        Kwater = Kwater/framecount;

        myoutfile.open("./AvePartitioncoeff.txt");
        myoutfile<<"# Number of frames analyzed: "<<framecount<<endl;
        myoutfile<<"# Klipid Kwater\n";
        myoutfile<<Ksurf<<"   "<<Kwater<<endl;
        myoutfile<<"# Energy barrier =-log(K) in [kT]:\n# (for water)  (for lipids)\n";
        myoutfile<<-log(Kwater)<<" "<<-log(Ksurf)<<endl;
        myoutfile.clear();
        myoutfile.close();

//Normalizing and wrting Phi distributions.
```

```
        ofstream outfile1,outfile2,outfile3;
        outfile1.open("AvePhi_slab_water.txt");
        outfile1<<"# Distribution of number fraction of water mol-les in the systen divided into many slabs alond Z-coordinate.";
        outfile1<<"\n# Numbers are averaged over "<<framecount<<" frames.\n# Format: Z phi";

        outfile2.open("AvePhi_slab_oil.txt");
        outfile2<<"# Distribution of number fraction of oil mol-les in the systen divided into many slabs alond Z-coordinate.";
        outfile2<<"\n# Numbers are averaged over "<<framecount<<"  frames.\n# Format: Z phi";

        outfile3.open("AvePhi_slab_lipid.txt");
        outfile3<<"# Distribution of number fraction of lipid mol-les in the systen divided into many slabs alond Z-coordinate.";
        outfile3<<"\n# Numbers are averaged over "<<framecount<<"  frames.\n# Format: Z phi";
        for (int liI=0;liI<Zslabs.size();liI++)
        {
                WATperslab(liI)=WATperslab(liI);//framecount;
                outfile1<<"\n "<< Zslabs(liI) <<" "<< WATperslab(liI)/framecount;

                Oilperslab(liI)=Oilperslab(liI);//framecount;
                outfile2<<"\n "<< Zslabs(liI) <<" "<< Oilperslab(liI)/framecount;

                Lipidperslab(liI)=Lipidperslab(liI);//framecount;
                outfile3<<"\n "<< Zslabs(liI) <<" "<< Lipidperslab(liI)/framecount;
        }
        outfile1.close();
        outfile2.close();
        outfile3.close();

}

void Averages::NormAndWrite_btwn(int HistCount, int framenumber, double LinkParam_ww, double LinkParam_ws)
{
        ofstream myoutfile;
        stringstream ss (stringstream::in | stringstream::out);
        char outname[200];
        int RDsize;
//Normalizing and Writing RD's
        for (int liI=0;liI<HistCount;liI++)
        {
                RDsize = allRDs[liI].RadialDistBins.size();
                ss.clear();
                ss<<"AVE"<<allRDs[liI].outname<<"RadialDist.txt";
                ss>>outname;

                cout<<"  Writing FINAL results to the file: "<<outname<<endl;
                myoutfile.open(outname);
                myoutfile<<"# Radial Distribution Function for selectors:"<<allRDs[liI].selector1<<"  &" <<allRDs[liI].selector2<<".\n# First Column Bins (distance),\n# Second Column is N(R)/(spher jacobian)\n";
                myoutfile<<"# Current frame:"<<framenumber<<"; Number of frames analyzed: "<<allRDs[liI].framnum<<endl;
                for (int liZ=0;liZ<RDsize;liZ++)
                {
                        if (isinf(allRDs[liI].RadialDistHist(liZ)))
                        cout<<"  INFINITE value of RadialDistHist("<<liZ<<")= "<<allRDs[liI].RadialDistHist(liZ)<<endl;
                        if (isnan(allRDs[liI].RadialDistHist(liZ)))
                        allRDs[liI].RadialDistHist(liZ) = 0;
                        //DEBUG +0.25 in bins to match Greg+Alex+Chris algorithm
                        myoutfile<<"\n "<<(allRDs[liI].RadialDistBins(liZ)+0.25)<<" "<<allRDs[liI].RadialDistHist(liZ)/allRDs[liI].framnum;
                }
                ss.clear();
                myoutfile.clear();
                myoutfile.close();

        }

//Normalizing and Writing Cluster Distribution.
```

```
        myoutfile.clear();
        myoutfile.open("AVEClustDist.txt");
        myoutfile<<"# Cluster Size Distribution Function.LinkParam = "<<LinkParam_ww<<",
            ↪ LinkParam_ws= "<<LinkParam_ws<<" [angst]\n# First Column Bines (number of molecules
            ↪ in cluster,\n# Second Column is normalized number of clusters with this number of
            ↪ molecules in them.";
        myoutfile<<"\n# Number of frames analyzed by now: "<<framenumber;
        int ClDsize = ClusterDistBins.size();

        for (int liI=0;liI<ClDsize;liI++)
        {
                myoutfile<<"\n "<<ClusterDistBins(liI)<<" "<<ClusterDistHist(liI)/framenumber;
        }
        myoutfile.clear();
        myoutfile.close();

//Normalizing and Writing 2D Cluster Distribution: WATER P(n)
        myoutfile.open("./AVE2DClustDist_Wat.txt");
        myoutfile<<"# Cluster Size Distribution Function.LinkParam ="<<LinkParam_ww<<"[angst]\n#
            ↪ First Column Bines (number of molecules in cluster,\n# Second Column is normalized
            ↪ number of clusters with this number of molecules in them.";
        myoutfile<<"\n# Number of frames analyzed by now: "<<clustframenum;
        myoutfile<<"# Number of water mol-s per one surfactant molecule WS_Cl_index = "<<WS_Cl_index
            ↪ /clustframenum<<endl;
        myoutfile<<"# Number of surfactant mol-s per one water molecues SW_Cl_index = "<<SW_Cl_index
            ↪ /clustframenum<<endl;
        int ClDsize = ClusterDistBins.size();

        for (int liI=0;liI<ClDsize;liI++)
        {
                myoutfile<<"\n "<<ClusterDistBins(liI)<<" "<<ClusterDistHist_Wat(liI)/clustframenum;
        }
        myoutfile.clear();
        myoutfile.close();

//Normalizing and Writing 2D Cluster Distribution: SURFACTANT P(m) for each n-water molecules in a
    ↪ cluster.
        //for (int liI=0;liI<ClDsize;liI++)
        //DEBUG: October 2009
        for (int liI=0;liI<10;liI++)
        {
                char name[50];
                ss.clear();
                ss<<"./AVE"<<ClusterDistBins(liI)<<"SurfClustDist.txt";
                ss>>name;
                myoutfile.open(name);
                myoutfile<<"# Cluster Size Distribution Function. ";
                myoutfile<<"\n# LinkParam_ws ="<<LinkParam_ws<<"[angst], LinkParam_ww = "<<
                    ↪ LinkParam_ww<<endl;
                myoutfile<<"# This gives the probability to a cluster to have certain number of
                    ↪ surfactants if the water number is "<<ClusterDistBins(liI)<<".\n# 1st Column:
                    ↪  number of molecules in cluster; 2nd Column: normalized probarility P_n_surf(
                    ↪ given m_water)";
                myoutfile<<"\n# Number of frames analyzed by now: "<<clustframenum;
                for (int liJ=0;liJ<ClDsize;liJ++)
                {
                        myoutfile<<"\n "<<ClusterDistBins(liJ)<<" "<<ClusterDistHist_Surf[liI](liJ)/
                            ↪ clustframenum;
                }
                myoutfile.clear();
                myoutfile.close();
        }

//Normalizing and Writing Partition coefficicients.

        myoutfile.open("./AVEpartitioncoeff.txt");
        myoutfile<<"# Number of frames: "<<framenumber<<endl;
        myoutfile<<"# Klipid  Kwater\n";
        myoutfile<<Ksurf/framenumber<<" "<<Kwater/framenumber<<endl;
        myoutfile<<"# Energy barrier =-log(K) in [kT]:\n# (for water)  (for lipids)\n";
        myoutfile<<-log(Kwater)<<" "<<-log(Ksurf)<<endl;
        myoutfile.clear();
```

```
        myoutfile.close();

//Normalizing and writing Phi distributions.

        ofstream outfile1,outfile2,outfile3;
        outfile1.open("./AVEPhi_slab_water.txt");
        outfile1<<"# Distribution of number fraction of water mol-les in the systen divided into
            ↪ many slabs alond Z-coordinate.";
        outfile1<<"\n# Numbers are averaged over "<<framenumber<<" frames.\n# Format: Z phi";

        outfile2.open("./AVEPhi_slab_oil.txt");
        outfile2<<"# Distribution of number fraction of oil mol-les in the systen divided into many
            ↪ slabs alond Z-coordinate.";
        outfile2<<"\n# Numbers are averaged over "<<framenumber<<"  frames.\n# Format: Z phi";

        outfile3.open("./AVEPhi_slab_lipid.txt");
        outfile3<<"# Distribution of number fraction of lipid mol-les in the systen divided into
            ↪ many slabs alond Z-coordinate.";
        outfile3<<"\n# Numbers are averaged over "<<framenumber<<"  frames.\n# Format: Z phi";
        for (int liI=0;liI<Zslabs.size();liI++)
        {
                outfile1<<"\n "<< Zslabs(liI) <<" "<< WATperslab(liI)/framenumber;

                outfile2<<"\n "<< Zslabs(liI) <<" "<< Oilperslab(liI)/framenumber;

                outfile3<<"\n "<< Zslabs(liI) <<" "<< Lipidperslab(liI)/framenumber;
        }
        outfile1.close();
        outfile2.close();
        outfile3.close();

}

void Averages::AveragePartCoeff(string filename)
{...}
```

These results were published in a peer reviewed journal as "*Nonequilibrium water transport in a nonionic microemulsion system*".