



UNIVERSITY OF BORÅS

SCHOOL OF ENGINEERING

COMPUTER SIMULATION OF PROTEIN
SUPERABSORBENTS

PREETISRI BASKARAN

This thesis comprises 30 ECTS credits and is a compulsory part in the Master of Science
with a Major in Resource recovery –Industrial Biotechnology, 120 ECTS credits
No. 7/2011

COMPUTER SIMULATIONS OF PROTEIN SUPERABSORBENTS

PREETISRI BASKARAN (bpreeti2002@gmail.com)

Master thesis

Subject Category: Technology

University of Borås
School of Engineering
SE-501 90 BORÅS
Telephone +46 033 435 4640

Examiner: Peter Ahlstrom

Supervisor,name: Edvin Erdtman

Supervisor,address: University of Borås, School of Engineering
501 90, Borås

Date: 27-10-2011

Keywords: Superabsorbents, C++, Monte Carlo, GEMMS, GROMACS

ACKNOWLEDGEMENT

I would like to owe my gratitude and thank my supervisor, Dr. Edvin Erdtman, Post Doc, School of Engineering, University of Borås, for his incredible support and patience to walk me through this work. His wide knowledge and his logical way of thinking have been of great value for me. His clear way of explaining the concepts no matter how long it takes, made me feel more confident. Your valuable contributions will always be treasured.

I would like to express my sincere gratitude to Dr. Tobias Gebäck, Post Doc, Department of Mathematics, University of Chalmers. His guidance helped me in all the time of my thesis work. He has made available his support in a number of ways even during his vacation period. I could not have imagined a better advisor and mentor for my thesis work.

I would also like to thank Dr. Peter Ahlström, Senior Lecturer, School of Engineering, University of Borås, who has supported me indirectly in many ways. He was the examiner of my thesis work who encouraged me continuously to finish my thesis work with good results.

Finally, I thank my family for encouraging me throughout my studies at University and supporting me both financially and morally with love and affection. Without their support and encouragement I would have never finished this degree.

ABSTRACT

The aim of this project is to develop superabsorbents from proteins in our case it is a zygomycetes fungus, where the process of this fungus is studied experimentally in University of Borås. As a result of this experiment by-products of protein are produced and this project is about the study to make use of such proteins as superabsorbing materials.

The water absorbing capacity is computationally studied using Gibbs ensemble Monte Carlo (GEMC) simulations to determine the absorbing properties and to effectively improve the absorbing capacity by using specific treatments, where this project focuses in using mesoscale force fields such as the MARTINI force field instead of atomistic force fields which were used in studying the structure of the superabsorbents.

For this purpose, the program code GEMMS is modified to make it read the desirable file formats in order to perform the simulations. C++ is used here to code the program to read the GROMACS topology file (.top) for MARTINI force field instead of, as currently reading the atom type file (.atp) and the residue type file (.rtp) for the AMBER99 atomistic force field.

Contents

1. Introduction	1
1.1. What is Superabsorbency?	2
1.2. A general overview about proteins	2
1.3. Superabsorbent polymers	3
2. Computational Methods	5
2.1. Atomistic VS coarse-grained simulations	5
2.2. Monte Carlo Simulation Method	6
2.2.1. Metropolis Method	6
2.2.2. General Approach of Monte Carlo Simulation	8
2.3. Ensembles	9
2.3.2 Periodic boundary conditions	9
2.4. Gibbs Ensemble Monte Carlo Simulations (GEMC)	10
2.4.1. Particle Displacement	11
2.4.2. Volume Change	11
2.4.3. Particle Exchange	12
2.5. Force field in general	13
2.5.1. Lennard-Jones potential	13
2.5.2. Buckingham potential	13
2.5.3. Ewald Summation	14
2.5.4 Bonded Potential	16
2.5.4.1 Bond stretching- Harmonic Potential	16
2.5.4.2 Harmonic angle potential	16
2.5.4.3 Improper and proper dihedrals	16
3. Software	17
3.1. GROMACS	17
3.2. GEMMS	17
3.3. Force field parameters	19
3.3.1. Atom type parameter (.atp)	19
3.3.2. Residue database (.rtp)	19
3.3.3. Topology file (.top)	20
3.3.4. Forcefield .itp file	22
4. C++	24
4.1 Why C++ in GEMMS?	24
5. Results and discussion	25
5.1 Future work	27
References	28
Appendix	30

1. Introduction

Hydrogels are defined as polymeric materials which show the capacity of swelling in water and retaining a significant fraction (>20%) of water within their structure, without dissolving in water.

Hydrogels that are generally referred to as superabsorbents are networks of polymer chains that are occasionally found as colloidal gels in which water is the dispersion medium. In other words, they are water absorbing natural or synthetic polymers containing over 99% of water. (Tönsing and Oldiges, 2001)

Superabsorbents have an excellent response to changing environment conditions such as temperature, pH, and solvent composition; they have been attracting many industrial applications (Buchholz et al., 1997). Because of their water retention property and subsequently, the slow release of water from swollen hydrogels, hydrogels are of significance as potential water retainer systems in the field of agriculture.

The aim of this project is to develop such superabsorbents from proteins. In our case it is the zygomycetes fungus taken as an example, where the process of this fungus is studied experimentally in University of Borås. (Mahdejabbari and Barghi, 2009) As a result of this experiment by-products of protein are produced. The absorption capability of these byproducts was studied by performing molecular simulations such as these byproducts could be used as bio-superabsorbents. For this purpose a software called GEMMS (Gibb's Ensemble Macro Molecular Simulation) was built in the University of Borås. (Gebäck, 2009, Errington and Panagiotopoulos, 2000, Johansson, 2007) and later it is modified in my thesis work in order to run simulations using Coarse grained models where currently it work with atomistic models.

Natural superabsorbent polymers are in general made of polysaccharides or proteins. These types of hydrogels are degraded by enzymatic reactions as well as chemical hydrolysis into smaller molecular weight products, consequently hydrogels prepared of natural resources experience degradation by decomposition in the backbone of the polymers (Pourjavadi et al., 2006). Protein superabsorbents in general could be an effective replacement of synthetic superabsorbents used these days as the water retention capacity is as good as that of the

synthetic superabsorbents. This kind of superabsorbents would reduce the excessive natural resource consumption and the destruction of the environment. Usually the protein superabsorbents use low cost protein sources like animal, plant and fungal proteins and also waste proteins from food industries.

The water absorbing capacity of these superabsorbents is computationally studied using the Gibbs Ensemble Monte Carlo (GEMC) simulations to determine the absorbing properties and to effectively improve the absorbing capacity by using specific treatments.

1.1. What is Superabsorbency?

Superabsorbency is defined as the swelling capability to absorb and retain aqueous solutions at least hundred times of the initial dry weight. (Buchholz et al., 1997) It is due to the electrostatic repulsion of the charges on the polymer chains and the osmotic pressure difference of the inner and the outer layer of the gels. (Ohmine and Tanaka, 1982) The most significant properties of such hydrogels is the swelling capacity and the elastic modulus of the respective hydrogel. These properties can decrease the swelling capacity with increasing cross link density. (Buchholz and Peppas, 1994)

1.2. A general overview about proteins

Proteins are biochemical compounds that consist of one or more polypeptides folded into a globular or fibrous form. There are about 20 different amino acids that occur naturally in proteins.

In protein molecules the amino acids are linked to each other by peptide bonds. Those occur between the amino group of one amino acid and carboxyl group of its neighbor. Figure 1 shows the long chain of proteins that have peptide linkage indicating the condensation of amino groups.

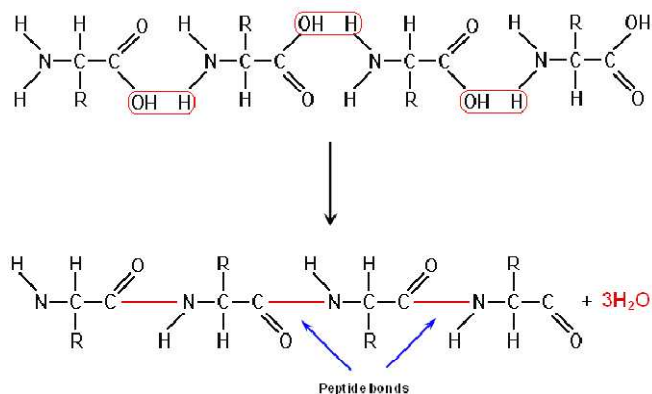


Figure1: Polypeptide Structure (Mahdejabbari and Barghi, 2009)

1.3. Superabsorbent polymers

Superabsorbent polymers are derived from either natural or synthetic sources. These superabsorbent polymers can be co-polymerized with hydrophilic polymers so that it can absorb a significant amount of water. The enzymatic reactions take place by breaking down or subsequent reduction to generate low molecular weight materials. On the whole, biodegradation can be defined as the conversion of materials into short fragments by enzymatic reactions. (Pourjavadi et al., 2006)

1.3.1. Ovalbumin

Ovalbumin (figure2) is a member of the water-soluble, heat coagulating, serpin super family of proteins. Albumins are present in plants and animal tissues, for example ovalbumin makes up to 65% of the total protein present in the egg white. (Bergmann and Niemann, 1937)

Ovalbumin consists of two chains with 385 amino acids, with a molecular weight of 45 kDa. Separation of albumins from the rest of the proteins is done by electrophoresis or by fractional precipitation. Table 1 presents the percentage of amino acid content in ovalbumin protein. (Bergmann and Niemann, 1937)

Previous work with protein based superabsorbents at University of Borås was to synthesize protein superabsorbent hydrogels from albumin protein by modifying with an acylating reagent and a bi-functional cross linker and to investigate the swelling behavior under specific conditions.(Mahdejabbari and Barghi, 2009) In their previous study at the university, the effect of certain physical and chemical parameters like protein structure, extent of

modification, protein concentration, pH, ionic strength, particle size and temperature were studied as the important factors for the water uptake behavior of superabsorbent hydrogels.

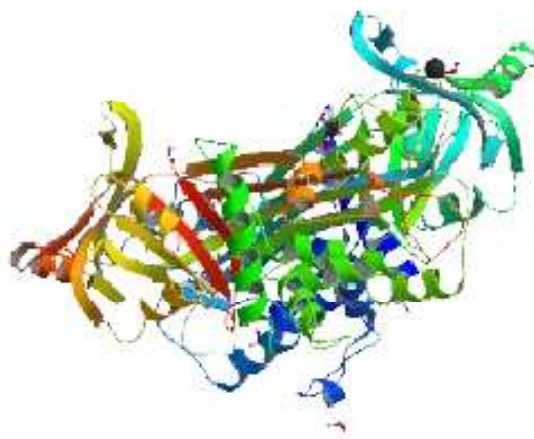


Figure 2: Ovalbumin Structure in cartoon format. The protein shows coloured α helices and β sheets (Majdejabbari and Barghii, 2009)

Table 1, Amino acid content of ovalbumin (Bergmann and Niemann, 1937)

Amino acid	Mol.wt	Percentage	Gm. molecule per 100 gr. protein	Polar	Non-polar
Glutamic	147.13	13.97	0.095	+	-
Cysteine	121.16	1.21	0.01	+	-
Lysine	146.188	4.97	0.034	+	-
Aspartic	133.10	5.98	0.045	+	-
Tyrosine	181.19	4.16	0.023	+	-
Methionine	149.21	5.07	0.034	-	+
Arginine	174.2	5.57	0.032	+	-
Histidine	155.16	1.39	0.009	-	+

2. Computational Methods

2.1. Atomistic VS coarse-grained simulations

An atomistic model of a protein contains all the atoms which are actually present in the protein, whereas the coarse grained model consists of so called grains which represent groups of atoms and the interactions are only defined between these grains (Müller-Plathe, 2002)

Atomistic simulations are mostly used in the studies of local structure and dynamics of bio-molecules and also to determine how a particular component can affect bio-molecule aggregate structure. In some force fields, the non-polar hydrogen atoms are neglected. These force fields are called united atom force fields. (Jue, 2010)

Coarse-grained (CG) simulations or mesoscale models can be used on length and timescales beyond atomistic capabilities. In many cases CG models have to be applied in order to attain to reach relevant size and time periods. In CG models the degrees of freedom is reduced by combining several atoms into one particle and eliminating short range dynamics. MARTINI model is one widely used CG model for lipids (Jue, 2010). In the case of MARTINI model, the dynamics are a factor of 4 faster than those of atomistic simulations and experiments. There are also models that are less detailed than MARTINI models, in those cases the water is not explicitly taken into account (Jue, 2010).

The MARTINI CG model is based on the four to one mapping, in which the average of four heavy atoms are represented by a single interaction center. The four main types of interaction sites are polar (P), nonpolar (N), apolar (C) and charged (Q). Each particle type has a number of subtypes, which allow for a more accurate representation of the chemical nature of the underlying atomic structure. In the main type, the subtypes are distinguished either by a letter denoting the hydrogen-bonding capabilities or by a number indicating the polarity (Marrink et al., 2007).

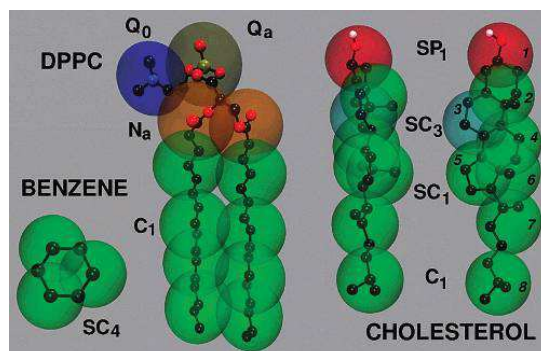


Figure 3: Example of CG bead types. The prefix “S” denotes a special class of CG sites introduced to model rings (Marrink et al., 2007)

2.2. Monte Carlo Simulation Method

The Monte Carlo (MC) simulation method was the first invented technique to perform a computer simulation and is therefore considered to be significant in the history of molecular modeling. Monte Carlo is used to refer the methods that use a technique known as *Importance Sampling*. This technique generates states of low energy. To estimate the efficiency of a MC sampling scheme, the sum of all accepted trial moves is divided by computing time. MC simulations have no momentum contribution opposed to the molecular dynamics simulation (Leach, 2001).

2.2.1. Metropolis Method

Metropolis MC simulation is a stochastic method which allows an efficient sampling of the multidimensional phase space of the system. The Metropolis method is often referred to as the Monte Carlo method and has become widely used in the simulation and the molecular modeling because evidently it leads to a faster convergence of the Markov chain (Leach, 2001).

The Markov chains of states are generated by metropolis method. Markov chain satisfies two conditions:

- 1) The outcome of every trial is depending upon the forthcoming trial and not the previous trials.
- 2) Each trial belongs to a possible finite set of outcomes (Leach, 2001).

A Monte Carlo algorithm consists of a group of Monte Carlo moves that generates a Markov chain of states. Consider a system in a state m , then the probability of moving this system from the state m to another state n can be defined as π_{mn} , where π is the transition matrix. Let us introduce a probability vector ρ . This vector defines the probability that the system is in a particular state. ρ_i is the probability of being in a state i . The equilibrium, limiting distribution of the Markov chain, ρ^* , is the result of applying the transition matrix an infinite number of times, $\rho^* = \lim_{N_{MC} \rightarrow \infty} \rho^{(0)} \pi^{N_{MC}}$, we get

$$\rho^* = \rho^* \pi \quad (1)$$

When simulating in canonical ensemble, ρ^* is reached when ρ_i is from to the Boltzmann factor. Boltzmann factor is the physical constant relating energy at the individual particle level with the temperature, it is given by the gas constant R divided by the Avogadro constant N_A , where $K = R / N_A$. Once the system has reached the equilibrium, any subsequent moves leave the system in equilibrium (Leach, 2001, Earl and Deem, 2008).

For the stricter condition of detailed balance to be satisfied, the net flux between two states m and n must be 0 at equilibrium, that is the rate of the transition from a state m to a state n equals the product of the population p_m and the appropriate element of the transition matrix π_{mn} , i.e.:

$$p_m \pi_{mn} = p_n \pi_{nm} \quad (2)$$

Considering the question whether the Monte Carlo move that is proposed could be accepted or rejected. The explanation is in the below text:

For this, consider the Metropolis acceptance criterion, the transition matrix between m and n can be given as,

$$\pi_{mn} = \alpha_{mn} p_{mn} \quad (3)$$

Here α_{mn} is the probability of proposing a move between the two states, p_{mn} is the probability of acceptance of the move.

$$\alpha_{mn} = \alpha_{nm} \quad (4)$$

Assuming the above equation as the case for most of the Monte Carlo moves, Metropolis proposed a scheme,

- If the new state n has a lower energy than the old state m then the move is accepted

$$\pi_{mn} = \alpha_{mn} (\rho_n \gtrsim \rho_m) \quad (5)$$

- Or, If the new state is higher in energy than the old state, the move is accepted with

$$\pi_{mn} = \alpha_{mn} (\rho_n / \rho_m) (\rho_n < \rho_m) \quad (6)$$

The above two condition apply if the initial and the final states m and n are different. If they are in the same state, then the transition matrix is given by,

$$\pi_{mm} = 1 - \sum_{n \neq m} \pi_{mn} \quad (7)$$

The overall metropolis criterion for the canonical ensemble can be summarized as,

$$p_{mn} = \min\{1, \exp(-\beta[u(n) - u(m)])\} \quad (8)$$

Here $\beta = 1/K_B T$ gives the reciprocal temperature with K_B as the Boltzmann constant and u denotes the potential energy.

2.2.2. General Approach of Monte Carlo Simulation

Here is a detailed procedure to demonstrate the validity of Monte Carlo algorithms (Frenkel and Smit, 2002),

- Decide the distribution of the sample we want, the distribution N depends on the details of the ensemble
- Impose the condition of detailed balance

$$K(m \rightarrow n) = K(n \rightarrow m) \quad (9)$$

Here $K(m \rightarrow n)$ is the flow of configuration from m to n .

$$K(m \rightarrow n) = N(m) \times \alpha(m \rightarrow n) \times \text{acc}(m \rightarrow n) \quad (10)$$

Here $N(m)$ defines the probability density to find a system in configuration m , $\text{acc}(m \rightarrow n)$ defines the acceptance probability of a move from m to n .

- The next step is to determine the probability of generating a particular configuration
- Finally the condition that needs to be fulfilled by acceptance rules is derived.

2.3. Ensembles

In the canonical ensemble, number of atoms (N), volume (V) and the temperature (T) are conserved during the simulation. In other words it is known as the constant temperature molecular dynamics or Monte Carlo. In the case of NVT, the energy of the endothermic and the exothermic reactions are exchanged with a thermostat.

In isothermal- isobaric ensemble number of atoms (N), pressure (P) and temperature (T) are conserved. Along with the thermostat, a barostat is needed. The barostat corresponds closely to the laboratory conditions with a flask open to ambient temperature and pressure. In NPT, during the simulation of biological membranes the isotropic pressure control is not appropriate. For example in lipid bilayers, the pressure control in the presence of constant membrane area (NPAT) or a constant surface tension “gamma” ($NP\gamma T$) (Leach, 2001). The Thermostats and Barostats are used only in Molecular Dynamics Simulations, In the case of MC, they are built into the algorithm.

2.3.2 Periodic boundary conditions

On simulating a real system, the particles from a macroscopic system is modeled such as it contains in the order of 10^{23} times less particles, that it is necessary to mimic a nearly infinite system in such a way that the system size effects are minimized, in this case most of the simulation is said to be carried via periodic boundary conditions. To avoid the particles interacting themselves, the particles are assumed only to interact with next image of each other particle, this is called the minimum image convention. The direct interaction between the particles are truncated at an even shorter radius in order to diminish the amount of calculations needed. The spherical truncation gives rise to less anisotropy than the minimum image truncation. When this truncation is done unexpectedly, the integration of the equations of motion will be done inaccurately in simulations due to fictitious step-function in the potential. By shifting the potential energy, the potential is made continuous,

$$V(r_{ij}) = \begin{cases} V_0(r_{ij}) - v_{cut} & r_{ij} \leq r_{cut} \\ 0 & r_{ij} > r_{cut} \end{cases} \quad (11)$$

Where r_{cut} is the truncation radius, V_0 is unshifted potential and $v_{cut} = V_0(r_{cut})$

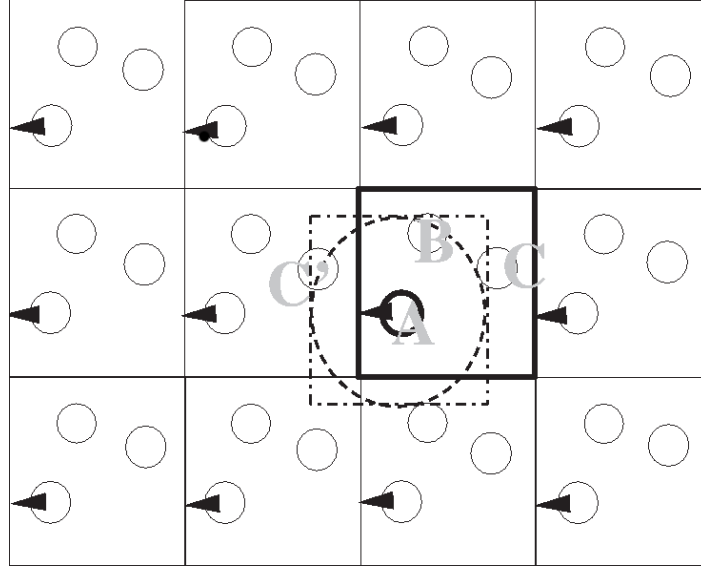


Figure 4: Periodic Boundary Conditions

The above image explains the periodic boundary conditions, where the particles leaving the central box on the left-hand side will enter it at the right hand side, like marked above.

Particle A with the minimum energy convention will interact with particle B and C' but not with particle C. If a spherical truncation radius is used it will only interact with particle B since the centre of particle C' is located outside the sphere. (Ahlström, 2007).

2.4. Gibbs Ensemble Monte Carlo Simulations (GEMC)

The Gibbs Ensemble Monte Carlo (GEMC) method is used to characterize phase transitions. GEMC simulations can be performed in NVT and NPT ensembles. The system contains molecules or atoms separated into two or more boxes. There is no physical border or surface between the boxes (Leach, 2001).

Generally three types of moves are possible (figure 4)

- 1) Displacement of particles within each box
- 2) Volume changes of the two boxes, by which the total volume of the two boxes remains constant (in the NVT ensemble)

3) Transfer of particles between the two boxes

The acceptance rules for these steps in the Gibbs ensemble is derived from the condition of the detailed balance in Eq.9.

2.4.1. Particle Displacement

The state n is obtained from state o through the displacement of a randomly selected particle in one of the boxes. The ratio of the statistical weights of the two configurations is derived as (Frenkel and Smit, 2002),

$$\frac{N(n)}{N(m)} = \frac{\exp[-\beta u(s_n^{n1})]}{\exp[-\beta u(s_m^{n1})]} \quad (12)$$

Here s is the coordinate, n is the new configuration and m is the old configuration by substituting this ratio into the condition of detailed balance,

$$acc(m \rightarrow n) = \min(1, \exp\{-\beta[u(s_n^{n1}) - u(s_m^{n1})]\}) \quad (13)$$

Where $acc(m \rightarrow n)$ denotes the acceptance probability of a move from m to n . The above equation is said to be the acceptance rule, which is identical to the conventional rule that is used in NVT ensemble simulation.

2.4.2. Volume Change

In the case of volume change, the ratio of the statistical weights before and after the move is given by the equation (Frenkel and Smit, 2002),

$$\frac{N(n)}{N(m)} = \frac{(V_1^n)^{n1} (V - V_1^n)^{N-n1}}{(V_1^m)^{n1} (V - V_1^m)^{N-n1}} \frac{\exp[-\beta u(s_n^N)]}{\exp[-\beta u(s_m^N)]} \quad (14)$$

Here V defines the total volume of the box and V_1 defines the volume of the box 1, the respective V_1^m and V_1^n defines the volume of the box 1 in the old and new configurations. The balance of this equation gives the acceptance rule for volume change,

$$acc(m \rightarrow n) = \min\left\{1, \frac{(V_1^n)^{n1} (V - V_1^n)^{N-n1}}{(V_1^m)^{n1} (V - V_1^m)^{N-n1}} \exp\{-\beta[u(s_n^N) - u(s_m^N)]\}\right\} \quad (15)$$

The above way of changing the volume was originally proposed by Panagiotopoulos et al (1989). Here u defines the potential energy of a configuration, N defines the number of particle and s defines the scaled coordinate of the respective particle (n or m).

2.4.3. Particle Exchange

In this case, a configuration n is generated from configuration m , where it is done by removing a particle from one box and inserting it into the other box. The ratio of the statistical weights of the configuration is (Frenkel and Smit, 2002):

$$\frac{N(n)}{N(m)} = \frac{n_1!(N-n_1)!V_1^{n_1-1}(V-V_1)^{N-(n_1-1)}}{(n_1-1)!(N-(n_1-1))!V_1^{n_1}(V-V_1)^{N-n_1}} \exp\{-\beta[u(s_n^N) - u(s_m^N)]\}$$

(16)

Here n_1 is the particle in box 1 and V_1 defines the volume of box 1.

Balancing this move, a new acceptance rule is made,

$$acc(m \rightarrow n) = \min\left\{1, \frac{n_1(V-V_1)}{(N-n_1+1)V_1} \exp\{-\beta[u(s_n^N) - u(s_m^N)]\}\right\}$$

(17)

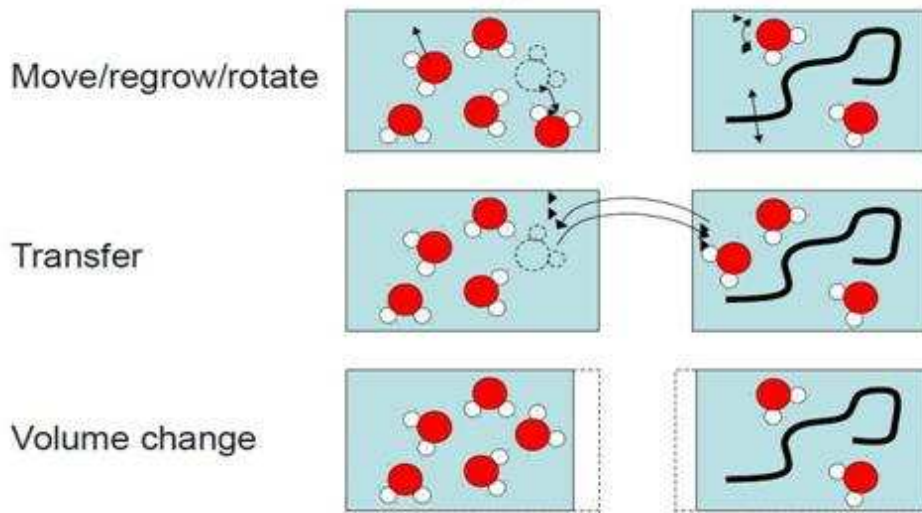


Figure 5. There are three types of moves, displacements within a box, transfer of water and volume change (figures by T. Gebäck)

2.5. Force field in general

2.5.1. Lennard-Jones potential

The Lennard-Jones (LJ) potential was first proposed by Mic in 1906 and then adopted by Lennard-Jones in the early 1920s (Lennard-Jones, 1934). This potential is also referred to as 6-12 potential. The LJ potential takes the form,

$$U_{LJ} = 4\epsilon \left(\frac{C_{ij}^{12}}{r_{ij}^{12}} - \frac{C_{ij}^6}{r_{ij}^6} \right) \quad (18)$$

where the parameters C_{ij}^{12} and C_{ij}^6 depend on pairs of atom types. Here i and j represent the atoms (Spoel et al., 2010).

When constructing the parameter matrix for the non-bonded LJ-parameters, two types of combination rules are used within GROMACS, the type 1 is given below:

$$C_{ij}^{(6)} = (C_{ii}^{(6)} C_{jj}^{(6)})^{\frac{1}{2}} \quad (19)$$

$$C_{ij}^{(12)} = (C_{ii}^{(12)} C_{jj}^{(12)})^{\frac{1}{2}}$$

An arithmetic average is used to calculate σ_{ij} , while the geometric average is used to calculate ϵ_{ij} (type 2) :

$$\sigma_{ij} = \frac{1}{2} (\sigma_{ii} + \sigma_{jj}) \quad (20)$$

$$\epsilon_{ij} = (\epsilon_{ii} \epsilon_{jj})^{1/2}$$

2.5.2. Buckingham potential

The Buckingham potential has more flexible and realistic repulsion than the LJ interaction. The drawback is that it is more expensive to compute (Spoel et al., 2010). The potential form of Buckingham potential is

$$V_{bh}(r_{ij}) = A_{ij} \exp(-B_{ij} r_{ij}) - \frac{C_{ij}}{r_{ij}^6} \quad (21)$$

There is only one set of combination rules for the Buckingham potentials between unlike atoms i and j ,

$$\begin{aligned}
A_{ij} &= (A_{ii}A_{jj})^{1/2} \\
B_{ij} &= \frac{1}{2}(B_{ii}B_{jj}) \\
C_{ij} &= (C_{ii}C_{jj})^{1/2}
\end{aligned} \tag{22}$$

2.5.3. Ewald Summation

Ewald summation was introduced as a technique in the early 1900's (Ewald, 1921) to sum the long range interactions between particles. The idea is to convert the potential energy of (eq.23), a single slowly converging sum into two quickly converging terms and a constant term (Spoel et al., 2010).

The total coulomb energy of a system with N particles and periodic images is given by,

$$V = \frac{f}{2} \sum_{n_x} \sum_{n_y} \sum_{n_{z^*}} \sum_i^N \sum_j^N \frac{q_i q_j}{r_{ij,n}} \tag{23}$$

Here n represents the periodic images. q_i is the charge of particle i. The star indicates that terms with $i = j$ should be omitted when $(n_x, n_y, n_z) = (0,0,0)$. $r_{ij,n}$ gives the real distance between the charges.

$$V = V_{dir} + V_{rec} + V_0 \tag{24}$$

The Ewald sum is written as the sum of these three parts (eq.23), namely the real space sum V_{dir} , the reciprocal sum V_{rec} and the constant term V_0 . Where,

$$V_{dir} = \frac{f}{2} \sum_{ij}^N \sum_{n_x} \sum_{n_y} \sum_{n_{z^*}} q_i q_j \frac{\text{erfc}(\beta r_{ij,n})}{r_{ij,n}} \tag{25}$$

$$V_{rec} = \frac{f}{2\pi V} \sum_{ij}^N q_i q_j \sum_{m_x} \sum_{m_y} \sum_{m_{z^*}} \frac{\exp(-((\frac{\pi m}{\beta})^2 + 2\pi i m(r_i - r_j))}{m^2} \tag{26}$$

$$V_0 = -\frac{f\beta}{\sqrt{\pi}} \sum_i^N q_i^2 \tag{27}$$

Here β is a parameter that determines the relative weight of the real and the reciprocal sums.

The feature of the Ewald's methods is that they split the original problem space into two problem spaces. The real or direct space part contains original point charges and the charge distributions centered at the same locations in space as each point charge, with the opposite charge and the reciprocal space contains negative of the charge distributions from the real space, by adding the two problem spaces the original point distribution is obtained.(Christopher and Cramer)

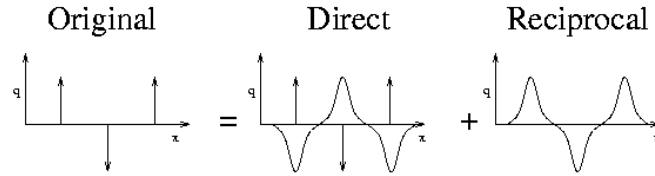


Figure 6: Representation of original, real and reciprocal problem spaces(Christopher and Cramer,2000).

Considering a N-particle system, the coulmb contribution is given by,

$$u_{coul} = \frac{1}{2} \sum_{i=1}^N q_i \varphi(r_i) \quad (28)$$

Where $\varphi(r_i)$ is the electrostatic potential at the position i :

$$\varphi(r_i) = \sum'_{j,n} \frac{q_j}{|r_{ij} + nL|} \quad (29)$$

Where n is the sum of overall periodic images and the overall particles is given by j where $j=i$ if $\mathbf{n}=0$. The contribution to the electrostatic potential at a point r_i due to a set of screened charges can be easily computed by direct summation, because the screened charge is a rapidly decaying function of r. Figure 5 demonstrates the three contributions of the electrostatic potential, first is the one due to the point charge q_i , second due to the screening charge cloud with the charge q_i and finally the compensating charge cloud with the charge q_i . In order to exclude the coulomb self interactions, all the three contributions should not be included at the position of the ion i. However it is convenient to retain the contribution due to the compensating charge distribution and correct the resulting interaction afterwards. The reason for retaining the compensating charge is that the charge distribution is not only a smoothly varying function but is also periodic, such a function can be represented by fourier series. (Frenkel and Smit, 2002).

2.5.4 Bonded Potential

Bonded interactions are based on a fixed list of atoms. There are bond stretching, bond angle and dihedral angle interactions and are explained below, (Spoel et al., 2010).

2.5.4.1 Bond stretching- Harmonic Potential

The bond stretching between two covalently bonded atoms i and j is represented by a harmonic potential,

$$V_b(r_{ij}) = \frac{1}{2} k_{ij}^b (r_{ij} - b_{ij})^2 \quad (30)$$

2.5.4.2 Harmonic angle potential

The bond angle vibration between a triplet of atoms i - j - k represented by a harmonic angle potential on the angle θ_{ijk} (Spoel et al., 2010).

$$V_a(\theta_{ijk}) = \frac{1}{2} k_{ijk}^\theta (\theta_{ijk} - \theta_{ijk}^0)^2 \quad (31)$$

2.5.4.3 Improper and proper dihedrals

Improper dihedrals are meant to keep planar groups planar or to prevent molecules from flipping over their mirror images,

$$V_{id}(\varepsilon_{ijkl}) = k_\varepsilon (\varepsilon_{ijkl} - \varepsilon_0)^2 \quad (32)$$

According to the IUPAC/IUB convention proper dihedrals is defined where φ is the angle between ijk and jkl planes, with zero corresponding to the cis configuration. The Ryckaert-Bellemans function for proper dihedrals is gives as,

$$V_{rb}(\varphi_{ijkl}) = \sum_{n=0}^5 C_n (\cos(\psi))^n \quad (33)$$

Where $\psi = \varphi - 180^\circ$ (Spoel et al., 2010).

3. Software

3.1. GROMACS

GROMACS (GRONingen Machine for Chemical Simulations) is a simulation package to run molecular dynamics simulations. It was originally developed in University of Groningen, and recently maintained in many places including University of Stockholm, University of Uppsala and Max Plank Institute for Polymer Research in Mainz. It is open source software.

The program is written to run in the operating systems UNIX and Linux. The program can run on multiple CPU cores. The package contains a program pdb2gmx, which is used to convert molecular coordinates from PDB file to the formats that are internally being used in GROMACS. Once this has been created the simulation can be run to make a trajectory file that describes the atom movements over a time. A notable use of GROMACS is that it is extensively used in protein folding (Spoel et al., 2010).

3.2. GEMMS

The GEMMS program (Gibbs Ensemble Macromolecule Simulation) is a program used for Gibbs Ensemble Monte Carlo simulations for macromolecules and solvents. A constant NVT or NPT is maintained for the simulation to run. The program is coded with C++ and Fortran 90 (Gebäck, 2009, Errington and Panagiotopoulos, 2000, Johansson, 2007).

The program reads a number of input files:

- Data file.
- Input file.
- Atom coordinate (.pdb) files
- State file
- Force field files

The data file consists of name patterns for reading and saving the other file types and also to specify the number of iterations and the location of the force field files.

The input file carries the force field parameters of small molecules such as water and ions. The iteration counts for equilibration and production periods, temperature, number of molecules, output intervals, and other options and parameters.

The atom coordinate files consist of the coordinates for macromolecules and also coordinate for small molecules. The file structure of the coordinate file is in the PDB format. The PDB file containing macromolecules gives details about the residue names and atom names compatible with the force field names of the atom type parameter (.atp) and the residue type parameter (.rtp) in the correct order.

The state file consists of a random generator seed and the parameters. This file is used to restart or continue a stopped simulation and to start simulation from the same state.

The Force field files will be discussed in chapter 3.3.

The Output Files from GEMMS are the following:

- results file (res.dat)
- output file (output.dat)
- plots files (plotdata.dat)
- standard output
- coordinate and state files (for continuing simulation)

The Result file specifies the information about equilibration period averages and the number of performed moves and the simulation time.

The output file consists of the information about the output and the progress of the simulation.

The plots file contains data for most of the state variables, in a TAB-separated format, which is easily read by Matlab, Octave, GNUPlot, etc.

The program also outputs some information to standard output (especially if the debug output level is high) - helpful for debugging.

The coordinate and state files are written in a format so that it could be used to restart the simulation from a saved configuration.

3.3. Force field parameters

In GEMMS force fields for macromolecules are read in a format supported by GROMACS, reading .itp, .atp and .rtp files. The force field used for proteins is Amber99. The TraPPE force field has also been used for polymers.

Force fields for small (solvent) molecules are read from the input file. For them only Lennard-Jones 12-6 interactions and Coulomb interactions (with Ewald summation) are used (Spoel et al., 2010)

3.3.1. Atom type parameter (.atp)

Each force field defines a set of atom types, that will have a name and mass (in atomic mass units). These listings are found in .atp files. Therefore it is the .atp files where the atom type can be changed or added. A sample of the .atp file is listed below.

```
amber99_1      79.90000      ; BR  bromine
amber99_2      12.01000      ; C   sp2 C carbonyl group
amber99_3      12.01000      ; CA  sp2 C pure aromatic (benzene)
amber99_4      12.01000      ; CB  sp2 aromatic C, 5&6 membered ring
junction
amber99_5      12.01000      ; CC  sp2 aromatic C, 5 memb. ring HIS
amber99_6      12.01000      ; CK  sp2 C 5 memb.ring in purines
amber99_7      12.01000      ; CM  sp2 C pyrimidines in pos. 5 & 6
amber99_8      12.01000      ; CN  sp2 C aromatic 5&6 memb.ring
junct.(TRP)
amber99_9      12.01000      ; CQ  sp2 C in 5 mem.ring of purines
between 2 N
amber99_10     12.01000      ; CR  sp2 arom as CQ but in HIS
amber99_11     12.01000      ; CT  sp3 aliphatic C
[...]
```

3.3.2. Residue database (.rtp)

The residue database file contains the information about bonds, charges, charge groups and improper dihedrals. The .rtp file is said to be a standard input file for the pdb2gmx and it is better not to change this file. The file structure is more similar to that of the .top file.

```

;tip3p
[ HOH ]

[ atoms ]
  OW  amber99_42  -0.834  0
  HW1 amber99_27  0.417  0
  HW2 amber99_27  0.417  0

[ bonds ]
  OW  HW1
  OW  HW2
[...]

[ dihedrals ]
  O      C      N1  H11  backbone_prop_2
  O      C      N1  H12  backbone_prop_2
  O      C      N2  H21  backbone_prop_2
  O      C      N2  H22  backbone_prop_2
  N1     C      N2  H21  backbone_prop_1
  N1     C      N2  H22  backbone_prop_1
  N2     C      N1  H11  backbone_prop_1
  N2     C      N1  H12  backbone_prop_1

[ impropers ]
  N1     N2     C      O  urea_improper_1
  C      H11    N1     H12
  C      H21    N2     H22
[...]

```

3.3.3 Topology file (.top)

The topology file is built following the GROMACS specification for a molecular topology. The *.top file is usually generated by the program pdb2gmx in the GROMACS interface (Spoel et al., 2010).

Below is a part of a topology file printed as an example:

```

[ moleculetype ]
; Name          nrexcl
Protein_chain_A  3

[ atoms ]
; nr          type  resnr  residue  atom  cgnr  charge  mass
typeB        chargeB  massB
; residue    1 NVAL  rtp  NVAL  q +1.0

```

```

      1      N3      1  NVAL      N      1      0.0577      14.01  ;
qtot 0.0577
      2      H      1  NVAL      H1     2      0.2272      1.008  ;
qtot 0.2849
      3      H      1  NVAL      H2     3      0.2272      1.008  ;
qtot 0.5121
      4      H      1  NVAL      H3     4      0.2272      1.008  ;
qtot 0.7393
      5      CT     1  NVAL      CA     5      -0.0054     12.01  ;
qtot 0.7339
      6      HP     1  NVAL      HA     6      0.1093      1.008  ;
qtot 0.8432
[...]
```

```

[ bonds ]
; ai    aj  funct      c0      c1      c2
c3
  1     2    1
  1     3    1
  1     4    1
  1     5    1
  5     6    1
  5     7    1
[...]
```

```

[ angles ]
; ai    aj    ak  funct      c0      c1      c2
c3
  2     1     3    1
  2     1     4    1
  2     1     5    1
  3     1     4    1
  3     1     5    1
  4     1     5    1
[...]
```

```

[ dihedrals ]
; ai    aj    ak    al  funct      c0      c1      c2
c3      c4      c5
  2     1     5     6    9
  2     1     5     7    9
  2     1     5    17    9
  3     1     5     6    9
  3     1     5     7    9
  3     1     5    17    9
[...]
```

```

[ dihedrals ]
; ai    aj    ak    al  funct      c0      c1      c2
c3
  5    19    17    18    4
  17   21    19    20    4
  21   40    38    39    4
  38   42    40    41    4
```

```
42    56    54    55    4
54    58    56    57    4
[...]
```

The explanation for the above file:

- The `molecule` type here gives the name of the molecule mentioned in the `.top` file (i.e.) it is the name of the protein that is used. The number in `nrexcl=3` is for excluding all the non-bonded interactions to atoms, that are (in this case) less than three bonds away
- The `atoms` section defines the type of molecule. The `nr` column provides with the individual atom number and type determines the type of atom. `resnr` gives the residue type number. Here residue refers to individual amino acid of a protein chain, where this amino acid is built up with a set of atoms. `Cgnr` divides groups of atoms into charged, which is unimportant for AMBER99 force field. `Charge` gives the charge of the atom and `mass` gives the atomic mass.
- For the `bonds` and `angles`, the function type is 1. The normal bond interactions, intended for harmonic potential have function type 1
- For `dihedrals`, in this case the function type of proper dihedrals is 9 (similar to type 1) and that of the improper dihedrals is 4 (similar to type 1). Here the type 1 is a periodic dihedral type (The function type 3 are intended for free energy calculations).

3.3.4. Forcefield .itp file

In the beginning of the file there is a row with the text `# include "forcefield.itp"`. This includes the bonded and non-bonded parameters. All the force field files are organized in the same way. Below is the `forcefield.itp` file from `amber99` force field printed as an example.

```
#define _FF_AMBER
#define _FF_AMBER99

[ defaults ]
; nbfunc      comb-rule      gen-pairs      fudgeLJ fudgeQQ
1             2             yes            0.5       0.8333
```

```
#include "ffnonbonded.itp"
#include "ffbonded.itp"
```

- `nbfunc` here is the non bonded function type .This non-bonded function type uses 1 (Lennard-Jones) or 2 (Buckingham)
- `comb-rule` is the combination rule which defines the parameters given for non – bonded interactions (Lennard - Jones or Buckingham potentials)
- `gen-pairs` is for pair generation, when it is set to ‘yes’ it generates 1-4 parameters that are not present in the pair list from normal Lennard-Jones parameters using `fudge LJ`
- `fudge LJ` is the factor to multiply LJ 1-4 interactions with default 1
- `fudge QQ` is the factor to multiply electrostatic 1-4 interactions with default 1

The `#include` is used in topologies to parse the data from another file. These files are force field specific. The `ffbonded.itp` contains all the bonded and `ffnonbonded.itp` contains all the non-bonded parameters

4. C++

A programming language is a set of instructions and a series of lexical conventions specifically designed to order computers what to do.

C++ is a general purpose language depending on the classes and the virtual functions to support object-oriented programming, templates to support generic programming, and providing low-level facilities to support detailed systems programming (Stroustrup, 1999).

4.1 Why C++ in GEMMS?

- The coding part of the program was written in C++ language. The C++ was coded to read the file formats such as the Topology (.top) and Force field (.itp) files.
- The C++ code could be compiled in almost any type of operating system without making any big changes, as it is the most used and ported programming language in the world. In this project, GEMMS works in the Linux.
- Code written in C++ is very short compared to other programming languages, because of the use of special characters is preferred to key words, saving some effort to the programmer
- The code from a C++ compilation is very efficient, due indeed to its duality as high-level and low-level language and to the reduced size of the language itself.
- An application's body in C++ can be made up of several source code files that are compiled separately and then linked together. This saves a lot of time, since it is not necessary to recompile the complete application when making a single change but only the file that contains it. In addition, this characteristic allows to link C++ code with code produced in other high level languages. GEMMS is a combined program. It is built with C++ and Fortran 90.
- On the whole, C++ is efficient for GEMMS.

5. Results and discussion

The C++ part of the program was modified successfully, so that instead of reading atomic type (.atp) and the residue type (.rtp) files, the GEMMS code now reads a topology file (.top).

The current GEMMS system is modified so that it could be made more compatible with the GROMACS version where the GROMACS files like .top and .itp could be used in the GEMMS for performing the simulation. Figure 6 describes the changes that were made to the GEMMS program. The figure to the left indicates the older version where the GEMMS used the .atp and .rtp file formats to run a simulation. The figure to the right indicates the new version, where instead of using the .atp and the .rtp files directly a pdb2gmx code was used in GROMACS to convert the .atp and .rtp files into a single .top file. This .top file was used in the GEMMS to run the simulation.

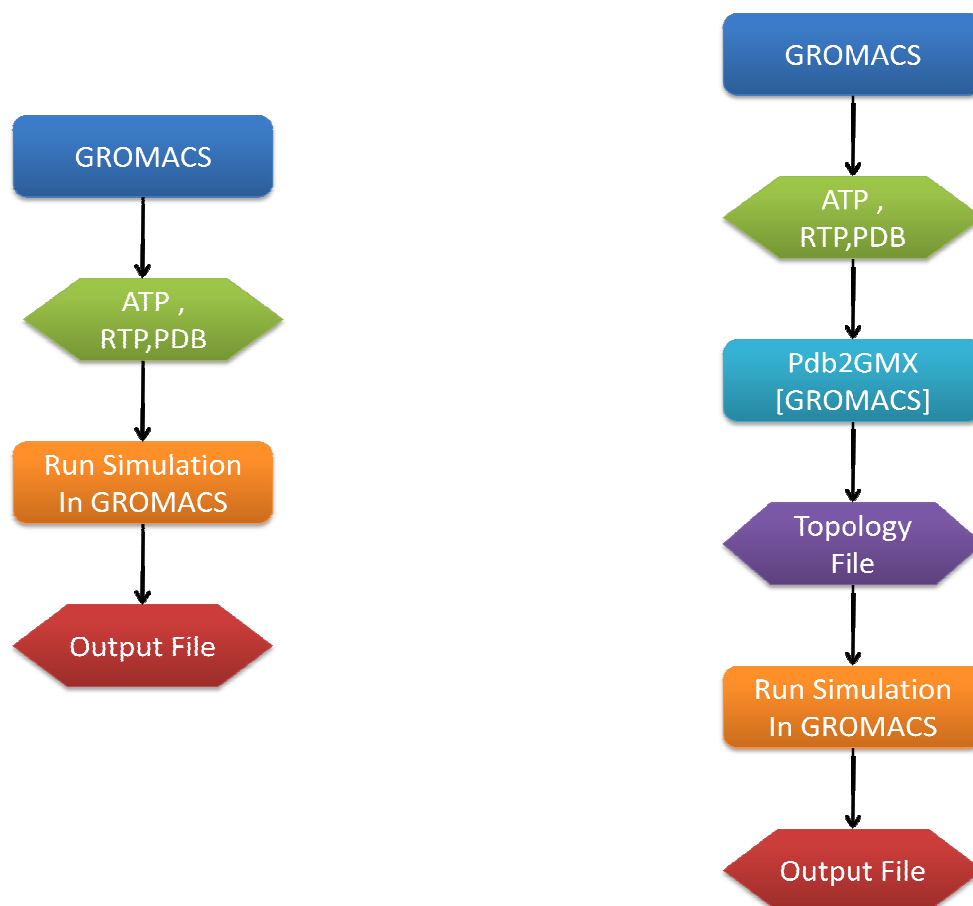


Figure 7: Flowchart about the files used in: (left) older version (right) new version

The major part of the modification in the program code is done in the Ffloader.cpp (see appendix), where a new function is created to read the .top file which replaced the functions that read the .atp and .rtp files. The top file consists of 4 sections; they are atoms, bonds, angles and dihedrals. To read these four sections, four sub functions were coded under the top function. Also in the beginning of the top files, the force field .itp file is included which means a #include code is being added in the beginning of the top file and so the respective code is written in C++ to read the .itp file section.

Few changes are done in the other files such as Energycomputer.cpp to modify the function types. In the previous version of GEMMS it was coded to read the function types for 1 and 3, for the respective itp files. But for this topology file the function types are 4 and 9, so the code has been modified to read the respective function types.

Finally in main.cpp changes in the parameter were done to adjust the modifications that were done in Ffloader.cpp

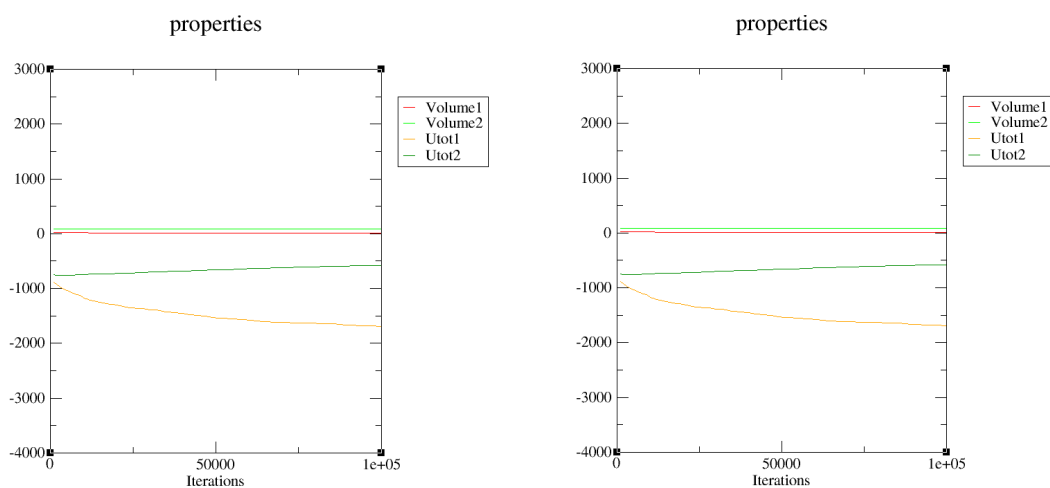


Figure 8: Simulation results of the GEMMS (left) older and the (right) newer version.

The simulation results of the older version and the newer version of GEMMS were compared, for which the results were same, as you can see in Figure 8 and the plotdata files below. The simulation was performed for atomistic model using amber99.

Plotdata results:

Old version:

Iter	Cycle	Volume1	Volume2	Pressure1	Pressure2	
RhoW1	RhoW2	Rho2	Nmol1	Nmol2	Utot1	
Ulj1	Ucou11	Uint1	Utot2	Ulj2	Ucou12	
Uint2	Ulj_sm-mm	Uljmm	Ucou1mm	mu1	mu2	
	501	1	19.68044	88.55903	-4038.496	-
897.4736	748.1987	78.15761	580.1445	492.2360	219.7640	
-863.7209	-82.23097	-781.6236	0.000000	-752.6539	-191.1677	
-937.7283	376.0697	-22.27006	-165.7581	1601.022	504.0769	
530.6515						
[...]						
4001	6	18.54387	86.87640	-3568.068	-804.9499	
781.3804	82.37712	594.0864	484.3780	227.6220	-1021.128	
-25.64500	-995.6112	0.000000	-755.3596	-186.3806	-968.3467	
399.1994	-22.73984	-164.0034	1594.250	-Infinity	385.5239	

New version:

Iter	Cycle	Volume1	Volume2	Pressure1	Pressure2	
RhoW1	RhoW2	Rho2	Nmol1	Nmol2	Utot1	
Ulj1	Ucou11	Uint1	Utot2	Ulj2	Ucou12	
Uint2	Ulj_sm-mm	Uljmm	Ucou1mm	mu1	mu2	
	501	1	19.68044	88.55903	-4038.496	-
897.4736	748.1987	78.15761	580.1445	492.2360	219.7640	
-863.7209	-82.23097	-781.6236	0.000000	-752.6539	-191.1677	
-937.7283	376.0697	-22.27006	-165.7581	1601.022	504.0769	
530.6515						
[...]						
4001	6	18.54387	86.87640	-3568.068	-804.9499	
781.3804	82.37712	594.0864	484.3780	227.6220	-1021.128	
-25.64500	-995.6112	0.000000	-755.3596	-186.3806	-968.3467	
399.1994	-22.73984	-164.0034	1594.250	-Infinity	385.5239	

5.1 Future work

This program should be tested with coarse grained simulations (i.e. with MARTINI force field) and the results should be checked. The reason for modifying the GEMMS system is that to make GEMMS to work for coarse grained simulations, where the simulation is performed on a larger system with less time consumption. Whereas in the atomistic models it takes a longer time in cases of simulating larger systems.

References

- AHLSTROM, P., Simulation and Modelling for Biotechnology 2007. 40-43
- BERGMANN, M. & NIEMANN, C. 1937. On the structure of proteins: Cattle hemoglobin, egg albumin, Cattle fibrin, and gelatin. *J. Biol. Chem*, 118, 301-314.
- BUCHHOLZ, F. L. & GRAHAM, A. T. 1997. *Modern Superabsorbent Polymer Technology*, John Wiley & sons, New York.
- BUCHHOLZ, F. L. & PEPPAS, N. A. 1994. *Superabsorbent Polymers, Science and Technology*, American Chemical Society, Washington D.C.
- CHRISTOPHER, D. & CRAMER, E. 2000. *The Development and Integration of a Distributed 3D FFT for a Cluster of Workstations* [Online]. Duke University. Available: http://www.usenix.org/publications/library/proceedings/als00/2000papers/papers/full_papers/cramer/cramer_html/ [Accessed 23-10-11 2011].
- EARL, D. J. & DEEM, M. W. 2008. Monte Carlo Simulations, *In: KUKOL, A. (ed.) Molecular Modeling of Proteins*, Humana Press, Totowa, New Jersey.
- ERRINGTON, J. R. & PANAGIOTOPOULOS, A. Z. 2000. *The Gibbs program* [Online]. Available: <http://kea.princeton.edu/jerring/gibbs> [Accessed 15-9-11].
- EWALD, P. P. 1921. Die berechnung optischer und elektrostatischer gitterpotentiale. *Ann. Phys.*, 369, 253-289.
- FRENKEL, D. & SMIT, B. 2002. *Understanding Molecular Simulation From Algorithm to Applications.*, Academic press, San Diego, California.
- GEBÄCK, T. 2009. GEMMS program. University of Borås.
- JOHANSSON, E. ET AL. 2007. *Monte Carlo simulations of equilibrium solubilities and structure of water in n-alkanes and polyethylene*. *J. Chem. Phys.* 126, 224-902.
- JUE, T. E. 2010. *Biomedical Applications of Biophysics*, Humana Press, Totowa, New Jersey.
- LEACH, A. R. 2001. *Molecular Modelling: Principles and Applications*, Pearson Education EMA, Essex, England.
- LENNARD-JONES, J. E. 1934. *On the Determination of Molecular Fields*, *Proc. R. Soc. Lond. A* **106** (738): 463-477.
- MAHDEJABBARI, S. & BARGHI, H. R. 2009. *Biosuperabsorbent from proteins*. Master Thesis, University Of Borås.
- MARRINK, S. J., RISSELADA, H. J., YEFIMOV, S., TIELEMAN, D. P. & VRIES, A. H. D. 2007. The MARTINI Force Field: Coarse Grained Model for Biomolecular Simulations. *J. Phys. Chem. B*, 111, 7812-7824.

- MÜLLER-PLATHE, F. 2002. Coarse-graining in polymer simulation: From the atomistic to the mesoscopic scale and back. *ChemPhysChem*, 3, 754-759.
- OHMINE, I. & TANAKA, T. 1982. Salt effects on the phase transition of ionic gels. *The Journal of Chemical Physics*, 77, 5725-5729.
- POURJAVADI, A., KURDTABAR, M., MAHDAVINIA, G. R. & HOSSEINZADEH, H. 2006. Synthesis and super-swelling behavior of a novel protein-based superabsorbent hydrogel, *Polymer Bulletin*, 57, 813-824.
- SPOEL, D. V. D., LINDAHL, E., HESS, B., BUUREN, A. R. V., APOL, E., MEULENHOF, P. J., TIELEMAN, D. P., SIJBERS, A. L. T. M., FEENSTRA, K. A., DRUNEN, R. V. & BERENDSEN, H. J. C. 2010. Gromacs User Manual version 4.5.4, www.gromacs.org [Accessed 15-09-11]
- STROUSTRUP, B. 1999. An Overview of the C++ Programming Language, AT&T Laboratories, Florham park, USA.
- TÖNSING, T. & OLDIGES, C. 2001. Molecular dynamic simulation study on structure of water in crosslinked poly(N-isopropylacrylamide) hydrogels *Phys. Chem. Chem. Phys.*, 3, 5542-5549.

Appendix

C++ code of GEMMS program, with separate files **FFloader.cpp**, **EnergyComputer.cpp** and **Main.cpp**. My contributions to the code are highlighted in bold and the existing code is shaded light.

FFloader.cpp

```
#include "FFloader.h"
#include <string>
#include <queue>
#include <stack>
#include <sstream>
#include <iostream>
#include <fstream>
#include <cstring>
#include <cstdlib>

using namespace std;

const double C_PI = 3.14159265358979;
const double DEG2RAD = C_PI / 180.0;

// 1000 / (kB*NA), converts from kJ/mol to K
const double kJpMOL_TO_K = 1e3 / 8.314472145136097;

#define MAX_LINELEN 256

FFloader::FFloader()
{
    m_debugmsg_level = 2;
}

void FFloader::PrintAAInfo(const string& name, const AAInfo &aai, ostream
&os)
{
    os << "Residue " << name << ":\n";
    os << aai.atoms.size() << " atoms:\n";
    for (NtoAAAtomInfoMap::const_iterator ait=aai.atoms.begin(); ait !=
aai.atoms.end(); ait++)
        os << ait->second.name << ": " << ait->second.charge << "\n";
    os << endl;
}

// Reads .atp file and fills m_atomtypes vector, m_atomindex
enum TOPSection
{
    TPL_NONE = 0, TPL_DEFAULTS, TPL_MOLECULETYPE, TPL_ATOMS, TPL_BONDS,
    TPL_ANGLES, TPL_DIHEDS
};

enum RTPSection
{
    RSN_NONE = 0, RSN_BONDEDTYPES = 1, RSN_MOLNONE, RSN_ATOMS, RSN_BONDS,
    RSN_EXCLUSIONS, RSN_ANGLES, RSN_DIHEDRALS, RSN_IMPROPERS
};
```

```

// clear the AAInfo structure
void ClearAAInfo(AAInfo &aai)
{
    aai.abbrev.clear();
    //aai.ctermdx = -1;
    //aai.ntermdx = -1;
}

// checks if the token definestr is defined in m_defines
bool FFloader::IsDefined(const std::string &definestr) const
{
    for (vector<StringDef>::const_iterator it = m_defines.begin();
it != m_defines.end(); it++) {
        if (it->token == definestr) return true;
    }
    return false;
}

// replaces all occurrences in str of defined strings (from m_defines)
void FFloader::ReplaceDefines(string &str) const
{
    for (vector<StringDef>::const_iterator it = m_defines.begin(); it !=
m_defines.end(); it++) {
        int pos = str.find(it->token);
        while (pos != string::npos) {
            if (isalnum(str[pos + it->token.length()]) || str[pos + it-
>token.length()] == '_') {
                pos += it->token.length();
            }
            else
                str.replace(pos, it->token.length(), it->replacement);
            pos = str.find(it->token, pos);
        }
    }
}

int FFloader::FindBondIndex(const string &abbrev1, const string &abbrev2)
const
{
    string teststr;

    teststr.reserve(10);
    teststr = abbrev1;
    teststr += ';';
    teststr += abbrev2;
    StoNMap::const_iterator snit = m_bondindex.find(teststr);
    if (snit != m_bondindex.end()) {
        return snit->second;
    }
    else {
        teststr = abbrev2;
        teststr += ';';
        teststr += abbrev1;
        snit = m_bondindex.find(teststr);
        if (snit != m_bondindex.end()) {
            return snit->second;
        }
    }
}

```

```

    }
    else {
        return -1;
    }
}
}

int FFloader::FindAngleIndex(const string &abbrev1, const string &abbrev2,
const string &abbrev3) const
{
    string teststr;

    teststr.reserve(15);
    teststr = abbrev1;
    teststr += ';';
    teststr += abbrev2;
    teststr += ';';
    teststr += abbrev3;
    StoNMap::const_iterator snit = m_angleindex.find(teststr);
    if (snit != m_angleindex.end()) {
        return snit->second;
    }
    else {
        teststr = abbrev3;
        teststr += ';';
        teststr += abbrev2;
        teststr += ';';
        teststr += abbrev1;
        snit = m_angleindex.find(teststr);
        if (snit != m_angleindex.end()) {
            return snit->second;
        }
        else {
            return -1;
        }
    }
}

int FFloader::LocateDihedral(const string &dihstr, int type) const
{
    StoNMap::const_iterator snit = m_dihedralindex.find(dihstr);
    if (snit != m_dihedralindex.end()) {
        if (type == -1 || m_dihedraltypes[snit->second].dtype == type)
            return snit->second;
    }
    return -1;
}

int FFloader::FindDihedralIndex(const string &abbrev1, const string
&abbrev2,
    const string &abbrev3, const string &abbrev4, int type) const
{
    char testbuf[20];
    int idx;
    string teststr;

    // check given order

```

```

    sprintf(testbuf, "%s;%s;%s;%s", abbrev1.c_str(), abbrev2.c_str(),
abbrev3.c_str(), abbrev4.c_str());
    teststr = testbuf;
    idx = LocateDihedral(teststr, type);
    if (idx >= 0)
        return idx;

    // check reverse order
    sprintf(testbuf, "%s;%s;%s;%s", abbrev4.c_str(), abbrev3.c_str(),
abbrev2.c_str(), abbrev1.c_str());
    teststr = testbuf;
    idx = LocateDihedral(teststr, type);
    if (idx >= 0)
        return idx;

    // check with wildcards at ends
    sprintf(testbuf, "%s;%s;%s;%s", "X", abbrev2.c_str(), abbrev3.c_str(),
"X");
    teststr = testbuf;
    idx = LocateDihedral(teststr, type);
    if (idx >= 0)
        return idx;

    // check reversed with wildcards at ends
    sprintf(testbuf, "%s;%s;%s;%s", "X", abbrev3.c_str(), abbrev2.c_str(),
"X");
    teststr = testbuf;
    idx = LocateDihedral(teststr, type);
    if (idx >= 0)
        return idx;

    // check with initial wildcard 'X'
    sprintf(testbuf, "%s;%s;%s;%s", "X", abbrev2.c_str(), abbrev3.c_str(),
abbrev4.c_str());
    teststr = testbuf;
    idx = LocateDihedral(teststr, type);
    if (idx >= 0)
        return idx;

    // check with double initial wildcard 'X'
    sprintf(testbuf, "%s;%s;%s;%s", "X", "X", abbrev3.c_str(),
abbrev4.c_str());
    teststr = testbuf;
    idx = LocateDihedral(teststr, type);
    if (idx >= 0)
        return idx;

    return -1;
}

```

```

void FFloader::LoadTOPFile(std::istream &is, std::ostream &resout, const
string &path)
{
    string line;
    string name, tmp;
    TOPSection curSection = TPL_NONE;
    vector<istream*> streamstack;
    vector<istream*>::iterator curstreamit;
    int nr=0, molnr=0;

```

```

m_molecules.clear();
m_molecules.reserve(3);

m_bondindex.clear();
m_angleindex.clear();
m_dihedralindex.clear();
m_bondtypes.clear();
m_angletypes.clear();
m_dihedraltypes.clear();

streamstack.push_back(&is);
curstreamit = streamstack.begin();
    resout << "Parsing TOP file..." << endl;

do {
    do {
        // read a line
        getline(**curstreamit, line);
        // remove comments, and replace defines
        int pos = line.find(';');
        if (pos != string::npos)
            line.resize(pos);
        ReplaceDefines(line);

        // parse line
        istringstream iss(line);
        switch (iss.peek()) {
            case '[': // new section
                iss.ignore(1);
                iss >> ws >> name;

                if (name == "moleculetype") curSection = TPL_MOLECULETYPE;
                    else if (name == "defaults") curSection =
TPL_DEFAULTS;
                else if (name == "atoms") curSection = TPL_ATOMS;
                else if (name == "bonds") curSection = TPL_BONDS;
                else if (name == "angles") curSection = TPL_ANGLES;
                else if (name == "dihedrals") curSection = TPL_DIHEDS;
                else {
                    if (m_debugmsg_level > 1)
                        resout << "Unknown section: " << name << endl;
                    curSection = TPL_NONE;
                }
                break;
            case '#': // preprocessor directive (include or define)
                iss.ignore(1);
                iss >> name;
                if (name == "define") {
                    StringDef def;
                    iss >> ws;
                    iss >> name;
                    def.token = name;
                    iss >> ws;
                    if (iss.eof()) {
                        def.replacement.clear();
                    }
                    else {
                        getline(iss, name);

```

```

        def.replacement = name;
    }
    m_defines.push_back(def);
    if (m_debugmsg_level > 3)
        resout << "Defining " << def.token << " as " <<
def.replacement << endl;
    }
    else if (name == "include") {
        getline(iss, tmp, '\\');
        getline(iss, tmp, '\\');
        tmp.insert(0, path);

        ifstream itis(tmp.c_str(), istream::in);
        if (itis.good()) {
            if (m_debugmsg_level > 0)
                resout << "Opening
ITP file:" << tmp << endl;
                LoadITPFile(itis, resout, path);
            }
            else {
                if (m_debugmsg_level > 1)
                    resout << "Error opening include file " <<
tmp << endl;
                }
            }
            else {
                if (m_debugmsg_level > 1)
                    resout << "Unknown preprocessor directive: " << name
<< endl;
                }
            break;
            default:
            iss >> ws;
            if (iss.eof())
                continue;
            switch (curSection)
            {
                case TPL_DEFAULTS:
                    iss >> m_itpparams.nbfunc >> m_itpparams.combrule;
                    iss >> tmp;
                    m_itpparams.pairgen = (tmp == "yes");
                    iss >> m_itpparams.fudgeLJ >> m_itpparams.fudgeQQ;
                    break;
                    case TPL_MOLECULETYPE:
                    {
                        molnr=m_molecules.size();//index of our
new molecule
                        nr=0;//reset atom number for this
molecule, used in reserve

                        Molecule mol;
                        m_molecules.push_back(mol);

                        m_molecules[molnr].m_aainfo.clear();

m_molecules[molnr].m_atomidxtoresidue.clear();
                        m_molecules[molnr].m_aabonds.clear();
                        m_molecules[molnr].m_aaangles.clear();
                        m_molecules[molnr].m_aadiheds.clear();

```

```

iss>>m_molecules[molnr].molname>>m_molecules[molnr].nrexcl;
        if (m_debugmsg_level > 3)
            resout << "Loading molecule "
<< m_molecules[molnr].molname << endl;
        break;
    }
    case TPL_ATOMS:
    {
        AAAAtomInfo aaainfo;
        string type, resname;
        int resnr, cgnr;//cgnr - is ignored
        iss >> nr >> aaainfo.type >> resnr >>
        resname >> aaainfo.name >> cgnr >> aaainfo.charge >>
aaainfo.mass;

        NtoAAInfoMap::iterator it =
m_molecules[molnr].m_aaainfo.find(resnr);
        if (it ==
m_molecules[molnr].m_aaainfo.end()) {
            //new residue, add it to our
map

std::pair<NtoAAInfoMap::iterator, bool> prInsert;
            AAInfo aai;

prInsert=m_molecules[molnr].m_aaainfo.insert(NtoAAInfoMap::value_type(resnr,
aai));

            it = prInsert.first;

            //initialize newly added

ClearAAInfo(it->second);
            it->second.abbrev=resname;

            if (m_debugmsg_level > 3)
                resout << "Added residue " << name << endl;
        }

        if (it !=
m_molecules[molnr].m_aaainfo.end()) {
            //find atome type in atomtypes
array

StoNMap::iterator
itAtomTypes=m_atomindex.find(aaainfo.type);
            if ( itAtomTypes !=
m_atomindex.end() ) {

aaainfo.idx=itAtomTypes->second;

m_atomtypes[aaainfo.idx].molweight = aaainfo.mass; // save mass for this
atom type

            }
            else
                if ( m_debugmsg_level > 1 )
                    resout << "Warning!

atom type:" << aaainfo.type <<

```

```

        " not found in atomtypes loaded from ITP file, missing
dependency" << endl;

                                NtoAAAtomInfoMap::value_type
val(nr, aaainfo);

std::pair<NtoAAAtomInfoMap::iterator, bool> prInsert;
                                prInsert=it-
>second.atoms.insert(val);
                                if (prInsert.second &&
m_debugmsg_level > 3)
                                resout << "Adding
atom " << aaainfo.name << " to residue " << resname << endl;
                                else
                                if (!prInsert.second) {
                                if (
m_debugmsg_level > 1)
                                resout
<< "Warning! atom nr " << nr << " already found in AAInfo collection" <<
endl;
                                break;
                                }
                                std::pair<NtoNMap::iterator,
bool> prLocalInsert;

prLocalInsert=m_molecules[molnr].m_atomidxtoresidue.insert(NtoNMap::value_t
ype(nr, resnr));
                                if (!prLocalInsert.second &&
m_debugmsg_level > 1)
                                resout << "Warning!
molecule nr " << molnr << " atom index " << nr
                                << "already exists in m_atomidxtoresidue" <<endl;
                                }
                                break;
                                }
                                case TPL_BONDS:
                                {
                                if (m_molecules[molnr].m_aabonds.size()
== 0)//1st time reserve size
                                m_molecules[molnr].m_aabonds.reserve(nr*1.5);//reserve approx size
                                int type;
                                AABond aabond;
                                aabond.params[0]=aabond.params[1]=0.0f;
                                iss >> aabond.lidx[0] >> aabond.lidx[1]
                                >> type;
                                aabond.ptype=type;//avoid reading ascii
                                type
                                iss >> ws;
                                if (iss.eof())
                                aabond.ptype = -
                                aabond.ptype;//no params found, negative value => we will need to get from
bondtypes data
                                else {
                                iss >> aabond.params[0] >>
aabond.params[1];

```

```

aabond.params[1] *=
kJpMOL_TO_K; // convert to K/nm^2
    }

    NtoNMap::iterator
itRes1=m_molecules[molnr].m_atomidxtoresidue.find(aabond.lidx[0]);
    NtoNMap::iterator
itRes2=m_molecules[molnr].m_atomidxtoresidue.find(aabond.lidx[1]);
    if (itRes1 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
    itRes2 ==
m_molecules[molnr].m_atomidxtoresidue.end()) {
        if (m_debugmsg_level > 1)
            resout << "Warning!
AABond atom indexes one or more not found. Molecule nr" << molnr
            << "
Atom Indexes " << aabond.lidx[0] << " " << aabond.lidx[1] << endl;
            break;
    }

m_molecules[molnr].m_aabonds.push_back(aabond);
    if (m_debugmsg_level > 3)
        resout << " Added bond " <<
aabond.lidx[0] << " " << aabond.lidx[1] << endl;
    break;
}
case TPL_ANGLES:
{
    if (m_molecules[molnr].m_aaangles.size()
== 0)//1st time reserve size

m_molecules[molnr].m_aaangles.reserve(nr*2);//reserve approx size
    AAAngle aaangle;
    int type;

aaangle.params[0]=aaangle.params[1]=0.0f;
    iss >> aaangle.lidx[0] >>
aaangle.lidx[1] >> aaangle.lidx[2] >> type;
aaangle.ptype=type;//avoid reading ascii
type

    iss >> ws;
    if (iss.eof())
        aaangle.ptype = -
aaangle.ptype; //no params found, negative value => we will need to get
from angletypes data

    else {
        iss >> aaangle.params[0] >>
aaangle.params[1];

        if (aaangle.ptype == 1) {
            aaangle.params[0]
aaangle.params[1]
        }
    }

    NtoNMap::iterator
itRes1=m_molecules[molnr].m_atomidxtoresidue.find(aaangle.lidx[0]);

```

```

                                NtoNMap::iterator
itRes2=m_molecules[molnr].m_atomidxtoresidue.find(aaangle.lidx[1]);
                                NtoNMap::iterator
itRes3=m_molecules[molnr].m_atomidxtoresidue.find(aaangle.lidx[2]);
                                if (itRes1 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
                                itRes2 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
                                itRes3 ==
m_molecules[molnr].m_atomidxtoresidue.end() ) {
                                if (m_debugmsg_level > 1)
                                        resout << "Warning!
AAngle atom indexes one or more not found. Molecule nr" << molnr
                                        << "
Atom Indexes " << aaangle.lidx[0] << " " << aaangle.lidx[1] << " "
                                        << "
aaangle.lidx[2] << endl;
                                        break;
                                }

m_molecules[molnr].m_aaangles.push_back(aaangle);
                                if (m_debugmsg_level > 3)
                                        resout << " Added Angle " <<
aaangle.lidx[0] << " " << aaangle.lidx[1]
                                        << " " <<
aaangle.lidx[2] << endl;
                                break;
                                }
                                case TPL_DIHEDS:
                                {
                                        if (m_molecules[molnr].m_aadiheds.size()
== 0)//1st time reserve size
                                        m_molecules[molnr].m_aadiheds.reserve(nr*3);//reserve approx
size
                                        AADihedral aadihed;
                                        int type, k;
                                        for(k=0; k<6 ; k++)
                                                aadihed.params[k]=0.0f;
                                        iss >> aadihed.lidx[0] >>
aadihed.lidx[1] >> aadihed.lidx[2] >> aadihed.lidx[3] >> type;
                                        aadihed.ptype=type;//avoid reading ascii
type
                                        iss >> ws;
                                        if (iss.eof())
                                                aadihed.ptype = -
aadihed.ptype;//no params found, negative number => we will need to get
params from dihedstype
                                        else {
                                                int nparams = (aadihed.ptype
== 1 ? 3 : 6);
                                                for (k = 0; k < nparams; k++)
                                                        iss >>
aadihed.params[k];
                                                if (aadihed.ptype == 1) {
                                                        aadihed.params[0]
*= DEG2RAD; // convert angle to rad

```

```

aadihed.params[1]
*= kJpMOL_TO_K; // convert to K
}
else if (aadihed.pptype == 3) {
    for (k=0;
k<nparams; k++)
aadihed.params[k] *= kJpMOL_TO_K; // convert to K
}
}
NtoNMap::iterator
itRes1=m_molecules[molnr].m_atomidxtoresidue.find(aadihed.lidx[0]);
NtoNMap::iterator
itRes2=m_molecules[molnr].m_atomidxtoresidue.find(aadihed.lidx[1]);
NtoNMap::iterator
itRes3=m_molecules[molnr].m_atomidxtoresidue.find(aadihed.lidx[2]);
NtoNMap::iterator
itRes4=m_molecules[molnr].m_atomidxtoresidue.find(aadihed.lidx[3]);
if (itRes1 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
itRes2 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
itRes3 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
itRes4 ==
m_molecules[molnr].m_atomidxtoresidue.end() ) {
    resout << "Warning!
DIHED atom indexes one or more not found. Molecule nr" << molnr
    << "
Atom Indexes " << aadihed.lidx[0] << " " << aadihed.lidx[1] << " "
    << "
aadihed.lidx[2] << " " << aadihed.lidx[3] << endl;
    break;
}

m_molecules[molnr].m_aadiheds.push_back(aadihed);
if (m_debugmsg_level > 3)
    resout << " Added DiHed " <<
aadihed.lidx[0] << " " << aadihed.lidx[1]
    << " " <<
aadihed.lidx[2] << " " << aadihed.lidx[3] << endl;
    break;
}
}
break;
}
} while (!(*curstreamit)->eof());

streamstack.erase(curstreamit);
if (streamstack.size() > 0)
    curstreamit = streamstack.end() - 1;
} while (streamstack.size() > 0);

if (m_debugmsg_level > 0) {
    resout << "Added " << m_molecules[molnr].m_aainfo.size() << "
residues.\n" << endl;
}

```

```

}

enum ITPSection
{
    ISN_NONE = 0, ISN_DEFAULTS=100, ISN_ATOMTYPES, ISN_BONDTYPES,
    ISN_ANGLETYPES,
    ISN_DIHEDTYPES,
};

// load atom types (LJ-parameters and abbreviations) to m_atomindex,
m_atomtypes
// load bond types to m_bondindex, m_bondtypes
// load angle types to m_angleindex, m_angletypes
// load dihedral types to m_dihedralindex, m_dihedraltypes

void FFloader::LoadITPFile(istream &is, ostream &resout, const string
&path)
{
    string line;
    string name, tmp;
    int curSection = ISN_NONE;
    vector<istream*> streamstack;
    vector<istream*>::iterator curstreamit;
    int nr=0, molnr=-1;
    bool bIgnoreTPLSections=false;

    streamstack.push_back(&is);
    curstreamit = streamstack.begin();

    if (m_debugmsg_level > 0)
        resout << "Parsing ITP file..."<< endl;

    do {
        do {
            // read a line
            getline(**curstreamit, line);
            // remove comments, and replace defines
            int pos = line.find(';');
            if (pos != string::npos)
                line.resize(pos);
            ReplaceDefines(line);

            // parse line
            istringstream iss(line);
            switch (iss.peek()) {
                case '[': // new section
                    {
                        iss.ignore(1);
                        iss >> ws >> name;
                        int nLen=name.length();
                        if (nLen > 0 && name[nLen-1] == ']')
                            name[nLen-1]='\0';

                        if (name == "defaults") curSection =
ISN_DEFAULTS;

                        else if (name == "atomtypes") {
                            curSection = ISN_ATOMTYPES;
                            bIgnoreTPLSections=true;//if atomtypes is found do not load TPL sections in
that file.
                        }
                    }
                }
            }
        } while (line != "");
    } while (curstreamit != streamstack.end());
}

```

```

        }
        else if (name == "bondtypes" )
curSection = ISN_BONDTYPES;
        else if (name == "angletypes")
curSection = ISN_ANGLETYPES;
        else if (name == "dihedraltypes")
curSection = ISN_DIHEDTYPES;
        else if (!bIgnoreTPLSections &&
strcmp(name.c_str(), "moleculetype") == 0) curSection = TPL_MOLECULETYPE;
        else if (!bIgnoreTPLSections &&
strcmp(name.c_str(), "atoms") == 0 ) curSection = TPL_ATOMS;
        else if (!bIgnoreTPLSections &&
strcmp(name.c_str(), "bonds") == 0 ) curSection = TPL_BONDS;
        else if (!bIgnoreTPLSections &&
strcmp(name.c_str(), "angles") == 0 ) curSection = TPL_ANGLES;
        else if (!bIgnoreTPLSections &&
strcmp(name.c_str(), "dihedrals") == 0) curSection = TPL_DIHEDS;
        else {
            if (m_debugmsg_level > 1)
                resout << "Unknown section: " << name <<
endl;
            curSection = ISN_NONE;
        }
        break;
    }
case '#': // preprocessor directive (include or define)
    iss.ignore(1);
    iss >> name;
    if (name == "define") {
        StringDef def;
        iss >> ws;
        iss >> name;
        def.token = name;
        iss >> ws;
        if (iss.eof()) {
            def.replacement.clear();
        }
        else {
            getline(iss, name);
            def.replacement = name;
        }
        m_defines.push_back(def);
        if (m_debugmsg_level > 3)
            resout << "Defining " << def.token << " as " <<
def.replacement << endl;
    }
    else if (name == "include") {
        istream* pis;
        getline(iss, tmp, '\\');
        getline(iss, tmp, '\\');
        tmp.insert(0, path);
        pis = new ifstream(tmp.c_str(), istream::in);
        if (pis->good()) {
            if (m_debugmsg_level > 2)
                resout << "Opening include file " << tmp
<< endl;
            streamstack.push_back(pis);
            curstreamit = streamstack.end() - 1;
        }
    }

```

```

        else {
            if (m_debugmsg_level > 1)
                resout << "Error opening include file " <<
tmp << endl;
        }
    }
    else {
        if (m_debugmsg_level > 1)
            resout << "Unknown preprocessor directive: " << name
<< endl;
    }
    break;

default:
    iss >> ws;
    if (iss.eof())
        continue;
    switch (curSection) {
        case ISN_DEFAULTS:
            iss >> m_itpparams.nbfunc >> m_itpparams.combrule;
            iss >> tmp;
            m_itpparams.pairgen = (tmp == "yes");

m_itpparams.fudgeLJ=m_itpparams.fudgeQQ=0.0f;
            iss >> m_itpparams.fudgeLJ >> m_itpparams.fudgeQQ;
            break;
        case ISN_ATOMTYPES:
            {
                float mass, charge; // these are ignored
                string ptype;
                iss >> name >> ws;

                    char ch=iss.peek();
                    if (ch <= '0' || ch >= '9')
                        iss >> tmp;
                    iss >> mass >> charge >> ptype;
                StonMap::iterator it = m_atomindex.find(name);
                if (it != m_atomindex.end()) {
                    AtomTypeData *p_atd = &m_atomtypes[it->second];
                    strncpy(p_atd->abbrev, tmp.c_str(), 4);
                    p_atd->abbrev[4] = '\\0';
                    iss >> p_atd->sigma >> p_atd->epsilon;

                        p_atd->epsilon *= kJpMOL_TO_K; // convert to K units

                            if (m_debugmsg_level > 3)

                                resout << "Modified atom type " << name <<
endl;
                }
            }
        else {
            AtomTypeData atd;
            strncpy(atd.abbrev, tmp.c_str(), 4);
            atd.abbrev[4] = '\\0';
            iss >> atd.sigma >> atd.epsilon;
            atd.epsilon *= kJpMOL_TO_K; // convert to K units
            atd.molweight = mass;
            m_atomindex.insert(StonPair(name, m_atomtypes.size()));
            m_atomtypes.push_back(atd);
            if (m_debugmsg_level > 3)

```

```

        resout << "Added atom type " << name << endl;
    }
    break;
}
case ISN_BONDTYPES:
{
    BondTypeData btd;
    int k, type;

    for (k = 0; k < 2; k++) {
        iss >> name;
        strncpy(btd.abbrev[k], name.c_str(), 4);
        btd.abbrev[k][4] = '\0';
    }
    iss >> type;
    btd.btype = type; // avoid reading ascii value
    if (btd.btype != 1 && btd.btype != 5) {
        if (m_debugmsg_level > 1)
            resout << "Warning! Bond " << btd.btype <<
" not supported!" << endl;
    }
    for (k = 0; k < 2; k++)
        iss >> btd.params[k];
        btd.params[1] *= kJpMOL_TO_K; //
convert to K/nm^2

    string key = btd.abbrev[0];
    key += ';';
    key += btd.abbrev[1];
    m_bondindex[key] = m_bondtypes.size();
    m_bondtypes.push_back(btd);
    if (m_debugmsg_level > 3)
        resout << "Added bond " << key << endl;
    break;
}

case ISN_ANGLETYPES:
{
    AngleTypeData atd;
    int k, type;

    for (k = 0; k < 3; k++) {
        iss >> name;
        strncpy(atd.abbrev[k], name.c_str(), 4);
        atd.abbrev[k][4] = '\0';
    }
    iss >> type;
    atd.atype = type; // avoid reading ascii value
    if (atd.atype != 1 && (m_debugmsg_level > 1))
        resout << "Warning! Angle " << atd.atype << " not
supported!" << endl;
    for (k = 0; k < 2; k++)
        iss >> atd.params[k];

    if (atd.atype == 1) {
        atd.params[0] *= DEG2RAD; // convert angle to radians

        atd.params[1] *= kJpMOL_TO_K; // convert to K/rad^2

```

```

}

string key = atd.abbrev[0];
for (k = 1; k < 3; k++) {
    key += ',';
    key += atd.abbrev[k];
}
m_angleindex[key] = m_angletypes.size();
m_angletypes.push_back(atd);
if (m_debugmsg_level > 3)
    resout << "Added angle " << key << endl;
break;
}
case ISN_DIHEDTYPES:
{
    DihedralTypeData dtd;
    int k, type;

    for (k = 0; k < 4; k++) {
        iss >> name;
        strncpy(dtd.abbrev[k], name.c_str(), 4);
        dtd.abbrev[k][4] = '\\0';
    }
    iss >> type;
    dtd.dtype = type; // avoid reading ascii value
    if (dtd.dtype != 1 && dtd.dtype != 3 && (m_debugmsg_level >
1))
        resout << "Warning! Dihedral " << dtd.dtype << " not
supported!" << endl;
    int nparams = (dtd.dtype == 1 ? 3 : 6);
    for (k = 0; k < nparams; k++)
        iss >> dtd.params[k];
    if (dtd.dtype == 1) {
        dtd.params[0] *= DEG2RAD; // convert angle to rad
        dtd.params[1] *= kJpMOL_TO_K; // convert to K
    }
    else if (dtd.dtype == 3) {
        for (k=0; k<nparams; k++)
            dtd.params[k] *= kJpMOL_TO_K; //
convert to K
    }

    string key = dtd.abbrev[0];
    for (k = 1; k < 4; k++) {
        key += ',';
        key += dtd.abbrev[k];
    }
    m_dihedralindex[key] = m_dihedraltypes.size();
    m_dihedraltypes.push_back(dtd);
    if (m_debugmsg_level > 3)
        resout << "Added dihedral " << key << endl;
    break;
}

case TPL_MOLECULETYPE:
{
    molnr=m_molecules.size();//index of our
new molecule

```

```

nr=0;//reset atom number for this
molecule, used in reserve

Molecule mol;
m_molecules.push_back(mol);

m_molecules[molnr].m_aaainfo.clear();

m_molecules[molnr].m_atomidxtoresidue.clear();
m_molecules[molnr].m_aabonds.clear();
m_molecules[molnr].m_aaangles.clear();
m_molecules[molnr].m_aadiheds.clear();

iss>>m_molecules[molnr].molname>>m_molecules[molnr].nrexcl;
if (m_debugmsg_level > 3)
    resout << "Loading molecule "
<< m_molecules[molnr].molname << endl;
break;
}

case TPL_ATOMS:
{
    AAAtomInfo aaainfo;
    string type, resname;
    int resnr, cgnr;//cgnr - is ignored
    iss >> nr >> aaainfo.type >> resnr >>
    resname >> aaainfo.name >> cgnr >> aaainfo.charge >>
aaainfo.mass;

    NtoAAInfoMap::iterator it =
m_molecules[molnr].m_aaainfo.find(resnr);
    if (it ==
m_molecules[molnr].m_aaainfo.end()) {
        //new residue, add it to our
map

std::pair<NtoAAInfoMap::iterator, bool> prInsert;
        AAInfo aai;

prInsert=m_molecules[molnr].m_aaainfo.insert(NtoAAInfoMap::value_type(resnr,
aai));

        it = prInsert.first;

        //initialize newly added

ClearAAInfo(it->second);
        it->second.abbrev=resname;

        if (m_debugmsg_level > 3)
            resout << "Added residue " << name << endl;
    }

    if (it !=
m_molecules[molnr].m_aaainfo.end()) {
        //find atome type in atomtypes
array

StoNMap::iterator
itAtomTypes=m_atomindex.find(aaainfo.type);
        if ( itAtomTypes !=
m_atomindex.end() ) {

```

```

aaainfo.idx=itAtomTypes->second;

m_atomtypes[aaainfo.idx].molweight = aaainfo.mass; // save mass of atom
for this atom type
    }
    else {
        if (
m_debugmsg_level > 1 )
            resout
            << "Warning! atom type:" << aaainfo.type <<
            " not
found in atomtypes loaded from ITP file, missing dependency" << endl;
        }
        NtoAAAtomInfoMap::value_type
val(nr, aaainfo);

std::pair<NtoAAAtomInfoMap::iterator, bool> prInsert;
    prInsert=it-
>second.atoms.insert(val);
        if (prInsert.second &&
m_debugmsg_level > 3)
            resout << "Adding
atom " << aaainfo.name << " to residue " << resname << endl;
            else
            if (!prInsert.second ) {
                if (
m_debugmsg_level > 1)
                    resout
                    << "Warning! atom nr " << nr << " already found in AAInfo collection" <<
                    endl;
                    break;
                }
                std::pair<NtoNMap::iterator,
bool> prLocalInsert;

prLocalInsert=m_molecules[molnr].m_atomidxtoresidue.insert(NtoNMap::value_t
ype(nr, resnr));
            if (!prLocalInsert.second &&
m_debugmsg_level > 1)
                resout << "Warning!
molecule nr " << molnr << " atom index " << nr
                << "already exists in m_atomidxtoresidue" <<endl;
                }
                break;
            }
            case TPL_BONDS:
            {
                if (m_molecules[molnr].m_aabonds.size()
== 0)//1st time reserve size

m_molecules[molnr].m_aabonds.reserve(nr*1.5);//reserve approx size
                int type;
                AABond aabond;
                aabond.params[0]=aabond.params[1]=0.0f;

```

```

iss >> aabond.lidx[0] >> aabond.lidx[1]
>> type;
aabond.ptype=type;//avoid reading ascii
type
iss >> ws;
if (iss.eof())
    aabond.ptype = -
aabond.ptype;//no params found, negative value => we will need to get from
bondtypes data
else {
    iss >> aabond.params[0] >>
aabond.params[1];
    aabond.params[1] *=
kJpMOL_TO_K; // convert to K/nm^2
}
NtoNMap::iterator
itRes1=m_molecules[molnr].m_atomidxtoresidue.find(aabond.lidx[0]);
NtoNMap::iterator
itRes2=m_molecules[molnr].m_atomidxtoresidue.find(aabond.lidx[1]);
if (itRes1 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
itRes2 ==
m_molecules[molnr].m_atomidxtoresidue.end()) {
    if (m_debugmsg_level > 1)
        resout << "Warning!
AABond atom indexes one or more not found. Molecule nr" << molnr
        << "
Atom Indexes " << aabond.lidx[0] << " " << aabond.lidx[1] << endl;
        break;
}

m_molecules[molnr].m_aabonds.push_back(aabond);
if (m_debugmsg_level > 3)
    resout << " Added bond " <<
aabond.lidx[0] << " " << aabond.lidx[1] << endl;
    break;
}
case TPL_ANGLE:
{
    if (m_molecules[molnr].m_aaangles.size()
== 0)//1st time reserve size
m_molecules[molnr].m_aaangles.reserve(nr*2);//reserve approx size
    AAAngle aaangle;
    int type;

aaangle.params[0]=aaangle.params[1]=0.0f;
iss >> aaangle.lidx[0] >>
aaangle.lidx[1] >> aaangle.lidx[2] >> type;
aaangle.ptype=type;//avoid reading ascii
type
iss >> ws;
if (iss.eof())
    aaangle.ptype = -
aaangle.ptype;//no params found, negative value => we will need to get from
angletypes data
else {

```

```

iss >> aaangle.params[0] >>
aaangle.params[1];
if (aaangle.ptype == 1) {
    aaangle.params[0]
    aaangle.params[1]
}
}
}
NtoNMap::iterator
itRes1=m_molecules[molnr].m_atomidxtoresidue.find(aaangle.lidx[0]);
NtoNMap::iterator
itRes2=m_molecules[molnr].m_atomidxtoresidue.find(aaangle.lidx[1]);
NtoNMap::iterator
itRes3=m_molecules[molnr].m_atomidxtoresidue.find(aaangle.lidx[2]);
if (itRes1 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
itRes2 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
itRes3 ==
m_molecules[molnr].m_atomidxtoresidue.end() ) {
    if (m_debugmsg_level > 1)
        resout << "Warning!
AAAngle atom indexes one or more not found. Molecule nr" << molnr << "
Atom Indexes " << aaangle.lidx[0] << " " << aaangle.lidx[1] << " " <<
aaangle.lidx[2] << endl;
        break;
}
}
m_molecules[molnr].m_aaangles.push_back(aaangle);
if (m_debugmsg_level > 3)
    resout << " Added Angle " <<
aaangle.lidx[0] << " " << aaangle.lidx[1] << " " <<
aaangle.lidx[2] << endl;
    break;
}
case TPL_DIHEDS:
{
    if (m_molecules[molnr].m_aadiheds.size()
== 0)//1st time reserve size
        m_molecules[molnr].m_aadiheds.reserve(nr*3);//reserve approx
size
        AADihedral aadihed;
        int type, k;
        for(k=0; k<6 ; k++)
            aadihed.params[k]=0.0f;
        iss >> aadihed.lidx[0] >>
aadihed.lidx[1] >> aadihed.lidx[2] >> aadihed.lidx[3] >> type;
        aadihed.ptype=type;//avoid reading ascii
type
        iss >> ws;
        if (iss.eof())

```

```

aadihed.ptype; //no params found, negative number => we will need to get
params from dihedstype
aadihed.ptype = -
else {
    int nparams = (aadihed.ptype
    == 1 ? 3 : 6);
    for (k = 0; k < nparams; k++)
        iss >>
aadihed.params[k];
    if (aadihed.ptype == 1) {
        aadihed.params[0]
        aadihed.params[1]
    }
    else if (aadihed.ptype == 3) {
        for (k=0;
k<nparams; k++)
aadihed.params[k] *= kJpMOL_TO_K; // convert to K
    }
}

NtoNMap::iterator
itRes1=m_molecules[molnr].m_atomidxtoresidue.find(aadihed.lidx[0]);
NtoNMap::iterator
itRes2=m_molecules[molnr].m_atomidxtoresidue.find(aadihed.lidx[1]);
NtoNMap::iterator
itRes3=m_molecules[molnr].m_atomidxtoresidue.find(aadihed.lidx[2]);
NtoNMap::iterator
itRes4=m_molecules[molnr].m_atomidxtoresidue.find(aadihed.lidx[3]);
if (itRes1 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
itRes2 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
itRes3 ==
m_molecules[molnr].m_atomidxtoresidue.end() ||
itRes4 ==
m_molecules[molnr].m_atomidxtoresidue.end() ) {
    resout << "Warning!
DIHED atom indexes one or more not found. Molecule nr" << molnr
    << "
Atom Indexes " << aadihed.lidx[0] << " " << aadihed.lidx[1] << " "
    <<
aadihed.lidx[2] << " " << aadihed.lidx[3] << endl;
    break;
}

m_molecules[molnr].m_aadiheds.push_back(aadihed);
if (m_debugmsg_level > 3)
    resout << " Added DiHed " <<
aadihed.lidx[0] << " " << aadihed.lidx[1]
    << " " <<
aadihed.lidx[2] << " " << aadihed.lidx[3] << endl;
    break;
}
}
break;

```

```

    }

    } while (!(*curstreamit)->eof());

    streamstack.erase(curstreamit);
    if (streamstack.size() > 0)
        curstreamit = streamstack.end() - 1;
    } while (streamstack.size() > 0);

    if (m_debugmsg_level > 0) {
        if (bIgnoreTPLSections)
            resout << "Added " << m_atomtypes.size() << " atom types, " <<
m_bondtypes.size() << " bond types, "
            << m_angletypes.size() << " angle types, " <<
m_dihedraltypes.size() << " dihedral types.\n" << endl;
        else
            if (molnr >= 0)//valid molecule nr was used above
                resout << "Added " <<
m_molecules[molnr].m_atomidxtoresidue.size() << " atoms, " <<
m_molecules[molnr].m_aainfo.size()
                    << " residues, " <<
m_molecules[molnr].m_aabonds.size() << " bonds, " <<
m_molecules[molnr].m_aaangles.size()
                    << " angles, " <<
m_molecules[molnr].m_aadiheds.size() << " dihedrals.\n" << endl;
            }
    }

}

// find local atom indices of neighbors to atom with local index atomidx

void FindBondedAtoms(const vector<AABond> &bonds, int atomidx, vector<int>
&neighbors)
{
    neighbors.clear();
    for (vector<AABond>::const_iterator bit = bonds.begin(); bit !=
bonds.end(); bit++) {
        if (bit->lidx[0] == atomidx)
            neighbors.push_back(bit->lidx[1]);
        else if (bit->lidx[1] == atomidx)
            neighbors.push_back(bit->lidx[0]);
    }
}

bool ContainsAngle(const vector<AAAngle> &angles, const AAngle &cura)
{
    for (vector<AAAngle>::const_iterator ait = angles.begin(); ait
!= angles.end(); ait++) {
        if (ait->lidx[1] == cura.lidx[1]) {
            if ((ait->lidx[0] == cura.lidx[0] && ait-
>lidx[2] == cura.lidx[2]) ||
                (ait->lidx[2] == cura.lidx[0] &&
ait->lidx[0] == cura.lidx[2]))
                {
                    return true;
                }
        }
    }
}

```

```

    }
    return false;
}

int FFloader::GenerateMolAngles(MoleculeData &mol, ostream &resout) const
{
    vector<int> nblast;
    AAAngle angle;
    for (int aidx = 0; aidx < mol.atoms.size(); aidx++) {
        FindBondedAtoms(mol.bonds, aidx, nblast);
        for (int nidx1=0; nidx1 < nblast.size(); nidx1++) {
            for (int nidx2=nidx1+1; nidx2 <
nblast.size(); nidx2++) {
                angle.lidx[0] = nblast[nidx1];
                angle.lidx[1] = aidx;
                angle.lidx[2] = nblast[nidx2];
                if (!ContainsAngle(mol.angles,
angle)) {
                    int saidx =
FindAngleIndex(m_ atomtypes[mol.atoms[angle.lidx[0]].idx].abbrev,
m_ atomtypes[mol.atoms[angle.lidx[1]].idx].abbrev,
m_ atomtypes[mol.atoms[angle.lidx[2]].idx].abbrev);
                    if (saidx >= 0) {
                        angle.ptype =
m_ angletypes[saidx].atype;
                        memcpy(angle.params, m_ angletypes[saidx].params, 2*sizeof(float));
                        mol.angles.push_back(angle);
                    }
                    else {
                        resout <<
"Warning! Could not find data for angle " << angle.lidx;
                    }
                }
            }
        }
    }
    return 0;
}

// checks if dih is contained in dvec (compares only indices, not
parameters)
bool ContainsDihedral(const vector<AADihedral> &dvec, const AADihedral
&dih)
{
    for (vector<AADihedral>::const_iterator pit = dvec.begin(); pit !=
dvec.end(); pit++) {
        int j;
        // compare in same order
        for (j = 0; j < 4; j++) {
            if (pit->lidx[j] != dih.lidx[j])
                break;
        }
        if (j == 4)
            return true;
    }
}

```

```

// compare mirrored
for (j = 0; j < 4; j++) {
    if (pit->lidx[j] != dih.lidx[3-j])
        break;
}
if (j==4)
    return true;
}
return false;
}

int FFloader::GenerateMolDihedrals(MoleculeData &mol, ostream &resout)
const
{
    vector<int> nblast1, nblast2;
    AADihedral dih;
    for (int aidx = 0; aidx < mol.atoms.size(); aidx++) {
        FindBondedAtoms(mol.bonds, aidx, nblast1);
        for (int nidx1=0; nidx1 < nblast1.size(); nidx1++) {
            FindBondedAtoms(mol.bonds, nblast1[nidx1],
nblast2);
            for (int nidx2=0; nidx2 < nblast1.size();
nidx2++) {
                if (nidx2 == nidx1) continue;
                for (int nidx3=0; nidx3 <
nblast2.size(); nidx3++) {
                    if (nblast2[nidx3] ==
aidx) continue;
                    dih.lidx[0] =
nblast1[nidx2];
                    dih.lidx[1] = aidx;
                    dih.lidx[2] =
nblast1[nidx1];
                    dih.lidx[3] =
nblast2[nidx3];
                    if
                    (!ContainsDihedral(mol.props, dih)) {
                        int saidx
= FindDihedralIndex(m_atomtypes[mol.atoms[dih.lidx[0]].idx].abbrev,
                    m_atomtypes[mol.atoms[dih.lidx[1]].idx].abbrev,
                    m_atomtypes[mol.atoms[dih.lidx[2]].idx].abbrev,
                    m_atomtypes[mol.atoms[dih.lidx[3]].idx].abbrev);
                        if (saidx
                    >= 0) {
                            dih.ptype = m_dihedraltypes[saidx].dtype;
                            memcpy(dih.params, m_dihedraltypes[saidx].params, 6*sizeof(float));
                            mol.props.push_back(dih);
                        }
                    }
                }
            }
        }
    }
}

```

```

else {
    resout << "Warning! Could not find data for dihedral " <<
dih.lidx;
}
}
}
}
}
return 0;
}

int FFloader::Load(const char* filestem)
{
    // read atom names and weights
    string filename(filestem);
    filename += ".top";
    int bsdidx = filename.rfind('\\');
    if (bsdidx == -1)
        bsdidx = filename.rfind('/');
    ifstream topis(filename.c_str(), ifstream::in);
    if (topis.fail())
        return 1;

    LoadTOPFile(topis, cout, filename.substr(0, bsdidx + 1));
    // GenerateAAAngles(cout);
    // GenerateAADihedrals(cout);

    if (m_debugmsg_level > 3) {
        for(int molnr=0;molnr<m_molecules.size(); molnr++){
            cout<< "Molecule " << m_molecules[molnr].molname <<
":\n";
            NtoAAInfoMap::iterator
it=m_molecules[molnr].m_aainfo.begin();
            for(int n=0;it != m_molecules[molnr].m_aainfo.end()
&& n < 2; it++, n++)
                PrintAAInfo(it->second.abbrev, it->second, cout);
        }
    }

    return 0;
}

// terminal is 0 for N-term, 1 for C-term, -1 otherwise
void TranslateAtomName(const string &pdbservice, string &ambername, int
terminal)
{
    if (pdbservice.compare(0,2,"HN") == 0) {
        ambername = "H";
        if (terminal == 0)
            ambername += pdbservice.substr(2);
    }
    else if (pdbservice.compare(0,2,"Cp") == 0)
        ambername = "C";
    else if (pdbservice.compare(0,2,"Op") == 0) {

```

```

    if ( terminal == 1)
        ambername = "OC1";
    else
        ambername = "O";
}
else if (pdbname.compare(0,2,"OC") == 0)
    ambername = "OC2";
else
    ambername = pdbname;
}

// convert bond aabond to bond with molecule-global atom indices (using
atomindexmap),
// replacing any indices to prev/next residue with proper indices from
// specialatomsmap.
// Converted bond is pushed onto molbonds vector.
// Any non-resolved bonds are pushed onto bqueue.
void FFloader::FillBond(std::map<int,int> &atomindexmap, const
std::vector<ModelAtom> &atoms,
                                                                    const
AABond &aabond, std::vector<AABond> &molbonds,  std::ostream &resout) const
{
    AABond curb;
    // replace indices into prev/next molecule
    for (int k = 0; k < 2; k++) {
        map<int, int>::iterator
itMolIdx=atomindexmap.find(aabond.lidx[k]);
        if (itMolIdx == atomindexmap.end()){
            if (m_debugmsg_level > 3)
                resout << "Warning! molecule
atom index not found for aabond index" << aabond.lidx[k] << endl;
            return;
        }
        curb.lidx[k] = itMolIdx->second;
    }
    if (aabond.pctype <= 0) {
        int idx = FindBondIndex(m_atomtypes[atoms[curb.lidx[0]].idx].abbrev,
m_atomtypes[atoms[curb.lidx[1]].idx].abbrev);
        if (idx >= 0) {
            curb.pctype = m_bondtypes[idx].btype;
            memcpy(curb.params, m_bondtypes[idx].params, 2*sizeof(float));
        }
        else
            curb.pctype = -1;
    }
    else {
        curb.pctype = aabond.pctype;
        memcpy(curb.params, aabond.params, 2 * sizeof (float));
    }
    molbonds.push_back(curb);
}

void FFloader::FillAngle(std::map<int,int> &atomindexmap, const
std::vector<ModelAtom> &atoms,
                                                                    const
AAAngle &aaangle, std::vector<AAAngle> &molangles,  std::ostream &resout)
const
{
    AAngle cura;

```

```

// replace indices into prev/next residue
for (int k = 0; k < 3; k++) {
    map<int, int>::iterator
itMolIdx=atomindexmap.find(aaangle.lidx[k]);
    if (itMolIdx == atomindexmap.end()){
        if (m_debugmsg_level > 3)
            resout << "Warning! molecule
atom index not found for aaangle index" << aaangle.lidx[k] << endl;
        return;
    }
    cura.lidx[k] = itMolIdx->second;
}
if (aaangle.ptype <= 0) {
    int idx = FindAngleIndex(m_atomtypes[atoms[cura.lidx[0]].idx].abbrev,
        m_atomtypes[atoms[cura.lidx[1]].idx].abbrev,
m_atomtypes[atoms[cura.lidx[2]].idx].abbrev);
    if (idx >= 0) {
        cura.ptype = m_angletypes[idx].atype;
        memcpy(cura.params, m_angletypes[idx].params, 2*sizeof(float));
    }
    else
        cura.ptype = -1;
}
else {
    cura.ptype = aaangle.ptype;
    memcpy(cura.params, aaangle.params, 2 * sizeof (float));
}
molangles.push_back(cura);
}

void FFloader::FillDihedral(std::map<int,int> &atomindexmap, const
std::vector<ModelAtom> &atoms,

    const AADihedral &aadih, std::vector<AADihedral> &moldih, int
dtype,

    std::ostream &resout) const
{
    AADihedral curd;
    // replace indices into prev/next molecule
    for (int k = 0; k < 4; k++) {
        map<int, int>::iterator itMolIdx=atomindexmap.find(aadih.lidx[k]);
        if (itMolIdx == atomindexmap.end()){
            if (m_debugmsg_level > 3)
                resout << "Warning! molecule
atom index not found for aadih index" << aadih.lidx[k] << endl;
            return;
        }
        curd.lidx[k] = itMolIdx->second;
    }
    if (aadih.ptype <= 0) {
        // parameters unknown, look for match in dihedraltypes
        int idx =
FindDihedralIndex(m_atomtypes[atoms[curd.lidx[0]].idx].abbrev,
            m_atomtypes[atoms[curd.lidx[1]].idx].abbrev,
m_atomtypes[atoms[curd.lidx[2]].idx].abbrev,
            m_atomtypes[atoms[curd.lidx[3]].idx].abbrev, dtype);
        if (idx >= 0) {
            curd.ptype = m_dihedraltypes[idx].dtype;

```

```

        memcpy(curd.params, m_dihedraltypes[idx].params, 6*sizeof(float));
    }
    else
        curd.ptype = -1;
}
else {
    // parameters already known, just copy
    curd.ptype = aadih.ptype;
    memcpy(curd.params, aadih.params, 6 * sizeof(float));
}
moldih.push_back(curd);
}

void FFloader::ConvertModelData(int chainnr, vector<PDBMolData> &pdbmols,
vector<PDBAtomData> &pdbatoms, ModelData &data, ostream &resout) const
{
    if (chainnr != (data.chains.size()+1)){//chainnr should be next
chain we will add
        if (m_debugmsg_level > 1)
            resout << "Warning! chain nr " << chainnr
<< " is out of sync with ModelData chain size "
<< data.chains.size() << ".
ConvertModelData failed" << endl;
            return;
        }
        if (chainnr > m_molecules.size()){//chainnr should be in bounds,
should match no. of molecules we loaded in TOP file
            if(m_debugmsg_level >1)
                resout << "Warning! chain nr " << chainnr
<< " is not found in TOP File molecules size "
<< m_molecules.size() << ".
ConvertModelData failed" << endl;
            return;
        }

        int cures, curatom=0, lastatom=0;
        MoleculeData *p_curmolecule;
        ModelResidue tmpres;
        string residuename;

        int atomcount=m_molecules[chainnr-1].m_atomidxtoresidue.size();
        ModelChain chain;
        data.chains.push_back(chain);
        data.chains[chainnr-1].molecules.reserve(pdbmols.size());
        residuename.reserve(5);

        for (vector<PDBMolData>::const_iterator molit = pdbmols.begin(); molit
!= pdbmols.end(); molit++) {
            // ToDo: These queues/maps not needed anymore in .top format
            // queues and maps for mapping to prev/next residue
            //map<int,int> forward_atomsmap, backward_atomsmap, lastbwd_atomsmap;
            //map<int,int> atomindexmap, prev_atomindexmap;
            //queue<AABond> bondqueue;
            //queue<AAAngle> anglequeue;
            //queue<AADihedral> propersqueue, impropersqueue;
            map<int,int> atomindexmap;
            p_curmolecule = new MoleculeData();

            p_curmolecule->residues.reserve(molit->nrresidues);

```



```

    }

    atom.charge = itAAAtom->second.charge;
    atom.idx = itAAAtom->second.idx;//Note: we dont have
any more atomtypedata collection, we will need to update ModelAtom for
this.
    memcpy(atom.x, pdbatoms[curatom].x, 3 * sizeof(double));
    strcpy(atom.pdbname, pdbatoms[curatom].name);
    p_curmolecule->atoms.push_back(atom);
}
}

//populate bonds, angles, dihedrs for this molecule
for(vector<AABond>::const_iterator bit=m_molecules[chainnr-
1].m_aabonds.begin());
    bit != m_molecules[chainnr-1].m_aabonds.end());
bit++)
    FillBond(atomindexmap, p_curmolecule->atoms, *bit,
p_curmolecule->bonds, resout);

    for(vector<AAAngle>::const_iterator ait=m_molecules[chainnr-
1].m_aaangles.begin());
    ait != m_molecules[chainnr-1].m_aaangles.end());
ait++)
    FillAngle(atomindexmap, p_curmolecule->atoms, *ait,
p_curmolecule->angles, resout);

    for(vector<AADihedral>::const_iterator dit=m_molecules[chainnr-
1].m_aadiheds.begin());
    dit != m_molecules[chainnr-1].m_aadiheds.end());
dit++)
    FillDihedral(atomindexmap, p_curmolecule->atoms, *dit,
p_curmolecule->propers, abs(dit->ptype), resout); // find dihedral matching
atoms and function type

    atomindexmap.clear();

    // add missing angles and dihedrals (spanning over multiple
residues)
    //GenerateMolAngles(*p_curmolecule, resout);
    //GenerateMolDihedrals(*p_curmolecule, resout);

    data.chains[chainnr-1].molecules.push_back(p_curmolecule);
}
}

```

```

int FFloader::LoadSystemFromPDB(const char *filename, ModelData &data)
const
{
    string line, tmp;
    vector<PDBAtomData> pdbatoms;
    vector<PDBMolData> pdbmolecules;
    int nr=0, lastres=0, resatoml=0, chainnr=0;
    PDBMolData curmolecule;
    PDBAtomData atomdata;
    bool bPrevAtomSOLorCL=false, bFirstAtomInChain=true;
    ifstream ispdb(filename);
    if (ispdb.fail())
        return 1;

```

```

if (m_debugmsg_level > 0) {
    cout << "Loading PDB file..."<<endl;
}

while (!ispdb.eof()) {
    // read a line
    getline(ispdb, line);

    // read command
    string cmd = line.substr(0, 6);
    istream iss(line);

    if (cmd.compare("REMARK") == 0) {
        continue;
    }
    else if (cmd.compare("CRYST1") == 0) {
        // get crystal lattice params
        iss.ignore(6);
        for (int k=0; k<6; k++)
            iss >> data.currentstate.crystalparams[k];
        for (int k=0; k<3; k++)
            data.currentstate.crystalparams[k] *= 0.1; //
convert distances to nm
        iss >> data.currentstate.crystaltype;
    }
    else if (cmd.compare(0,5,"MODEL") == 0) {
        // new molecule
        curmolecule.nratoms = 0;
        curmolecule.nrresidues = 0;
        curmolecule.atperres.clear();
        curmolecule.atperres.reserve(50);
        if ((pdbatoms.capacity() - pdbatoms.size()) < 100)
            pdbatoms.reserve(pdbatoms.size() + 100);

        if (m_debugmsg_level > 4)
            cout << "New model:" << endl;
    }
    else if (cmd.compare(0,4,"ATOM") == 0) {
        // one atom
        iss.ignore(6);
        iss >> nr; // atom order number
        iss >> atomdata.name;
        iss >> atomdata.resname;
        iss.ignore(2); // ignore residue type
        iss >> atomdata.resnr;
        iss >> atomdata.x[0] >> atomdata.x[1] >> atomdata.x[2];
        for (int k=0; k<3; k++)
            atomdata.x[k] *= 0.1; // convert to nm

        bool bReachedSOLorCL=( strcmp(atomdata.resname, "SOL") == 0 ||
            strcmp(atomdata.resname, "CL") == 0 );
        if ( lastres > atomdata.resnr || (!bPrevAtomSOLorCL &&
bReachedSOLorCL) )//if resnr reset, we are doing a new molecule
        {
            //save off previous molecule
            curmolecule.nratoms = nr - 1;//last read atom nr
for the prev residue

```

```

        curmolecule.nrresidues = lastres;
        curmolecule.atperres.push_back(curmolecule.nratoms
- resatom1 + 1);

        resatom1 = nr;
        lastres = atomdata.resnr;

        pdbmolecules.push_back(curmolecule);

        //reset values for new molecule
        curmolecule.nratoms = 0;
        curmolecule.nrresidues = 0;
        curmolecule.atperres.clear();
        curmolecule.atperres.reserve(50);
        if ( (pdbatoms.capacity() - pdbatoms.size()) < 100
)
                pdbatoms.reserve(pdbatoms.size() + 100);
    }

    if ( bReachedSOLorCL )
    {
        if(!bPrevAtomSOLorCL && m_debugmsg_level > 0)
            cout<< "Reached SOL or CL residue name:"
<< atomdata.resname << " at nr:"<< nr << " skipped all the rest of
residues" << endl;
        bPrevAtomSOLorCL=bReachedSOLorCL;
        continue;//skip SOL or CL atoms till the next TER
    }
    else
    {
        bPrevAtomSOLorCL=false;
    }

    pdbatoms.push_back(atomdata);

    if (bFirstAtomInChain) {
        resatom1 = nr;
        bFirstAtomInChain=false;
    }
    else
    if (lastres != atomdata.resnr) {
        curmolecule.atperres.push_back(nr - resatom1);
        resatom1 = nr;
    }
    lastres = atomdata.resnr;

    if (m_debugmsg_level > 4)
        cout << "Read atom " << atomdata.name << " in residue " <<
atomdata.resname << endl;

}
else if (cmd.compare(0,3,"TER") == 0) {
    // terminate atom list
    if (!bPrevAtomSOLorCL) {
        curmolecule.nratoms = nr;//last read atom nr
        curmolecule.nrresidues = lastres;//last read resnr
        curmolecule.atperres.push_back(curmolecule.nratoms - resatom1 + 1);
        pdbmolecules.push_back(curmolecule);
    }
}

```

```

//reset values for new molecule
curmolecule.nratoms = 0;
curmolecule.nrresidues = 0;
curmolecule.atperres.clear();
curmolecule.atperres.reserve(50);
if ( (pdbatoms.capacity() - pdbatoms.size()) < 100 )
    pdbatoms.reserve(pdbatoms.size() + 100);

bPrevAtomSOlorCL=false;bFirstAtomInChain=true;
chainnr++;
nr=0;lastres=0;resatoml=0;

if (m_debugmsg_level > 0) {
    cout << "Loaded Chain" << chainnr << ":" <<
pdbmolecules.size() << " molecules with "
        << pdbatoms.size() << " atoms from PDB
file." << endl;
}
ConvertModelData(chainnr, pdbmolecules, pdbatoms, data, cout);

pdbmolecules.clear();
pdbatoms.clear();
data.chains[chainnr-1].neighbor_ignoredist =
m_molecules[chainnr-1].nrexcl;

// compute neighbor matrices
for (vector<MoleculeData*>::iterator mit =
data.chains[chainnr-1].molecules.begin();
    mit != data.chains[chainnr-1].molecules.end();
mit++)
    (*mit)->ComputeNeighborMatrix();

if (m_debugmsg_level > 4) {
    int i=0;
    cout << "Molecule nr " << i+1 << ":" << endl;
    cout << "Atoms:" << endl;
    for (vector<ModelAtom>::iterator it =
data.chains[chainnr-1].molecules[i]->atoms.begin();
        it != data.chains[chainnr-
1].molecules[i]->atoms.end(); it++)
        cout << i << " " << m_atomtipes[it-
>idx].abbrev << endl;
    cout << "Bonds:" << endl;
    for (vector<AABond>::iterator it =
data.chains[chainnr-1].molecules[i]->bonds.begin();
        it != data.chains[chainnr-
1].molecules[i]->bonds.end(); it++)
        cout << '\t' << it->lidx[0] << "," << it-
>lidx[1] << "," << endl;
    cout << "Angles:" << endl;
    for (vector<AAAngle>::iterator it =
data.chains[chainnr-1].molecules[i]->angles.begin();
        it != data.chains[chainnr-
1].molecules[i]->angles.end(); it++)
        cout << '\t' << it->lidx[0] << "," << it-
>lidx[1] << "," << it->lidx[2] << endl;
    cout << "Propers:" << endl;
    for (vector<AADihedral>::iterator it =
data.chains[chainnr-1].molecules[i]->propers.begin();

```

```

        it != data.chains[chainnr-
1].molecules[i]->propers.end()); it++)
        cout << '\t' << it->lidx[0] << "," << it-
>lidx[1] << "," << it->lidx[2] << "," << it->lidx[3] << endl;
        cout << "Improper:" << endl;
        for (vector<AADihedral>::iterator it =
data.chains[chainnr-1].molecules[i]->impropers.begin();
        it != data.chains[chainnr-
1].molecules[i]->impropers.end()); it++)
        cout << '\t' << it->lidx[0] << "," << it-
>lidx[1] << "," << it->lidx[2] << "," << it->lidx[3] << endl;
    }
    }
    else if (cmd.compare("ENDMDL") == 0) {
        // end molecule
    }
}

if (m_debugmsg_level > 0) {
    cout << "Loaded " << chainnr << " chains." <<endl;
    cout << "Converted PDB data." << endl;
}

data.atomtypes = m_atomtypes; // copy atom type data
data.parameters.fudgeLJ = m_itpparams.fudgeLJ;
data.parameters.fudgeQQ = m_itpparams.fudgeQQ;
data.parameters.neighbor_ignoredist = 3; // default value, not found in
TOP-file

return 0;
}

```

Energy Computerc.cpp

```

#include "EnergyComputer.h"
#include <cmath>
#include <iostream>

using namespace std;

const double C_PI = 3.14159265358979324;

template<class T> T sqr(const T &a) { return a*a; }

// Complementary error function from Numerical Recipes.
double my_erfc(double x)
{
    double t, z, val;

    z = fabs(x);
    t = 1.0 / (1.0 + 0.5 * z);

    val = t * exp(-z * z - 1.26551223 + t *
        (1.00002368 + t * (0.37409196 + t *

```

```

        (0.09678418 + t * (-0.18628806 + t *
        (0.27886807 + t * (-1.13520398 + t *
        (1.48851587 + t * (-0.82215223 + t * 0.17087277))))))));

    if (x < 0.0)
        return 2.0 - val;
    else
        return val;
}

// distance between two points
double ptdist(const double x1[], const double x2[])
{
    return sqrt(sqr(x1[0]-x2[0]) + sqr(x1[1]-x2[1]) + sqr(x1[2]-x2[2]));
}

// squared distance between two points
double ptdist2(const double x1[], const double x2[])
{
    return (sqr(x1[0]-x2[0]) + sqr(x1[1]-x2[1]) + sqr(x1[2]-x2[2]));
}

// scalar product
double dot(const double x1[], const double x2[])
{
    return (x1[0]*x2[0] + x1[1]*x2[1] + x1[2]*x2[2]);
}

void vdiff(const double v1[], const double v2[], double res[])
{
    res[0] = v1[0] - v2[0];
    res[1] = v1[1] - v2[1];
    res[2] = v1[2] - v2[2];
}

// angle between two vectors
double vangle(const double v1[], const double v2[])
{
    return acos(dot(v1,v2)/sqrt(dot(v1,v1)*dot(v2,v2)));
}

// cross product v1 x v2
void vcross(const double v1[], const double v2[], double res[])
{
    res[0] = v1[1]*v2[2] - v1[2]*v2[1];
    res[1] = v1[2]*v2[0] - v1[0]*v2[2];
    res[2] = v1[0]*v2[1] - v1[1]*v2[0];
}

// maps the distance r to nearest distance in periodic system
// with periodicity boxdim[0:2]
// assuming r is between points in at most next to nearest boxes
void mapdist_to_nearest(double r[], const double boxdim[])
{

```

```

for (int k=0; k<3; k++) {
    if (r[k]>0.0) {
        if (2.0*r[k] > boxdim[k])
            r[k] -= boxdim[k];
        }
    else {
        if (-2.0*r[k] > boxdim[k])
            r[k] += boxdim[k];
        }
    }
}

// computes LJ-potential with sharp cutoff rcutoff
double EnergyComputer::LennardJones(double r[], double sigma, double
eps, double rcutoff) const
{
    double r2 = dot(r,r);
    if (r2 <= sqr(rcutoff)) {
        double sor2x = sigma * sigma / r2;
        double sor6x = sor2x*sor2x*sor2x;
        return 4.0 * eps * (sor6x*sor6x - sor6x);
    }
    else
        return 0.0;
}

// computes tail correction for cutoff LJ-potential
// see eq. (3.2.5) in Frenkel-Smit (2nd ed)
// Ndensity is [# LJ-atoms per volume]
double EnergyComputer::LennardJonesCorr(double rcutoff, double eps,
double sigma, double Ndensity) const
{
    double sor = sigma/rcutoff;
    double sor3 = sor*sor*sor;
    return 8.0/3.0*C_PI * Ndensity * eps * (sigma*sigma*sigma) *
(sor3*sor3*sor3/3.0 - sor);
}

// computes the contribution from one charge pair q1,q2 to the real
part of the Ewald sum
// see Eq. (12.1.24) in Frenkel-Smit (2nd ed)
// sqrtalpha is alpha^(1/2) by the definition from Frenkel-Smit
double EnergyComputer::EwaldReal(double q1, double q2, double
sqrtalpha, double r, double rcutoff) const
{
    if (r <= rcutoff)
        return q1*q2*my_erfc(sqrtalpha*r) / r;
    else
        return 0.0;
}

double EnergyComputer::BondEnergy(const AABond &bond, const
vector<ModelAtom> &atoms) const
{
    if (bond.ptype == 1) {

```

```

        double dist = ptdist(atoms[bond.lidx[0]].x, atoms[bond.lidx[1]].x);
        double en = 0.5 * bond.params[1] * sqr(dist - bond.params[0]);
        //cout << "Bond type 1: " << bond.lidx[0] << ' ' << bond.lidx[1] <<
": " << en << endl;
        return en;
    }
    else {
        cerr << "Invalid bond type " << (int)bond.ptype << ": " <<
bond.lidx[0] << ", " << bond.lidx[1] << endl;
        return 0.0;
    }
}

```

```

double EnergyComputer::AngleEnergy(const AAAngle &angle, const
vector<ModelAtom> &atoms) const
{
    if (angle.ptype == 1) {
        double v1[3], v2[3];
        vdiff(atoms[angle.lidx[0]].x, atoms[angle.lidx[1]].x, v1);
        vdiff(atoms[angle.lidx[2]].x, atoms[angle.lidx[1]].x, v2);
        double theta = vangle(v1,v2);
        double en = 0.5 * angle.params[1] * sqr(theta - angle.params[0]);
        //cout << "Angle type 1: " << en << endl;
        return en;
    }
    else if (angle.ptype == 2) {
        double v1[3], v2[3];
        vdiff(atoms[angle.lidx[0]].x, atoms[angle.lidx[1]].x, v1);
        vdiff(atoms[angle.lidx[2]].x, atoms[angle.lidx[1]].x, v2);
        double theta = vangle(v1,v2);
        double en = 0.5 * angle.params[1] * sqr(theta - angle.params[0]);
        //cout << "Angle type 2: " << en << endl;
        return en;
    }
    else {
        cerr << "Invalid angle type " << (int)angle.ptype << ": "
<< angle.lidx[0] << ", " << angle.lidx[1] << ", " <<
angle.lidx[2] << endl;
        return 0.0;
    }
}

```

```

double EnergyComputer::DihedralEnergy(const AADihedral &dih, const
vector<ModelAtom> &atoms)
{
    double v1[3],v2[3],n1[3],n2[3];

    // Compute angle phi between (i,j,k) and (j,k,l) planes (phi=0 for
cis config (IUPAC/IUB convention))
    vdiff(atoms[dih.lidx[0]].x, atoms[dih.lidx[1]].x, v1); // j->i vector
    vdiff(atoms[dih.lidx[2]].x, atoms[dih.lidx[1]].x, v2); // j->k vector
    vcross(v1,v2,n1);
    vdiff(atoms[dih.lidx[1]].x, atoms[dih.lidx[2]].x, v1); // k->j vector
    vdiff(atoms[dih.lidx[3]].x, atoms[dih.lidx[2]].x, v2); // k->l vector
    vcross(v1,v2,n2);

```

```

    if (dih.ptype == 1) {
        // periodic type (eq. (4.57) in GROMACS manual)
        double phi = vangle(n1,n2);
        double en = dih.params[1] * (1.0 + cos(dih.params[2]*phi -
dih.params[0]));
        m_dih1E += en;
        /*cout << "Dihedral type 1: " << dih.lidx[0] << "," << dih.lidx[1]
<< ","
                << dih.lidx[2] << "," << dih.lidx[3] << ": " << en <<
endl;*/
        return en;
    }
    else if (dih.ptype == 3) {
        // Ryckaert-Bellemans function (eq. (4.58) in GROMACS manual)
        double cosphi = dot(n1,n2)/sqrt(dot(n1,n1)*dot(n2,n2));
        double phifactor = 1.0;
        double sum = 0.0;
        for (int k=0; k<6; k++) {
            sum += dih.params[k] * phifactor;
            phifactor *= -cosphi; // minus because of different angle
convention
        }
        m_dih3E += sum;
        /*cout << "Dihedral type 3: " << dih.lidx[0] << "," << dih.lidx[1]
<< ","
                << dih.lidx[2] << "," << dih.lidx[3] << ": " << sum <<
endl;*/
        return sum;
    }
    else if (dih.ptype == 2) {
        double v1[3], v2[3];
        vdiff(atoms[dih.lidx[0]].x, atoms[dih.lidx[1]].x, v1);
        vdiff(atoms[dih.lidx[2]].x, atoms[dih.lidx[1]].x, v2);
        double theta = vangle(v1,v2);
        return 0.5 * dih.params[1] * sqr(theta - dih.params[0]);
    }
    else if (dih.ptype == 4) {
        double v1[3], v2[3];
        vdiff(atoms[dih.lidx[0]].x, atoms[dih.lidx[1]].x, v1);
        vdiff(atoms[dih.lidx[2]].x, atoms[dih.lidx[1]].x, v2);
        double theta = vangle(v1,v2);
        return 0.5 * dih.params[1] * sqr(theta - dih.params[0]);
    }
    else {
        cerr << "Invalid dihedral type " << (int)dih.ptype << ": "
                << dih.lidx[0] << "," << dih.lidx[1] << ","
                << dih.lidx[2] << "," << dih.lidx[3] << endl;
        return 0.0;
    }
}

```

```

// computes the internal LJ potential in a molecule, ignoring 3 nearest
neighbors
double EnergyComputer::InternalLJEnergy(const MoleculeData &moldata,

```

```

        const ModelParams &params, const ModelState &state, const
vector<AtomTypeData> &atomtypes) const
{
    double ljenergy = 0.0;
    double sigmac, epsc, r[3];
    vector<ModelAtom>::const_iterator ait, ait2;

    // compute Lennard-Jones contribution
    for (ait = moldata.atoms.begin(); ait != moldata.atoms.end(); ait++)
    {
        for (ait2 = ait+1; ait2 != moldata.atoms.end(); ait2++) {
            int dist = moldata.NeighborDistance(ait->idx, ait2->idx);
            if (dist < params.neighbor_ignoredist) // ignore nearest
neighbors
                continue;

            // combination rules for epsilon, sigma
            sigmac = (atomtypes[ait->idx].sigma + atomtypes[ait2->idx].sigma)
/ 2.0;
            epsc = sqrt(atomtypes[ait->idx].epsilon * atomtypes[ait2-
>idx].epsilon);

            // compute distance and map periodically
            vdiff(ait->x, ait2->x, r);
            mapdist_to_nearest(r, state.crystalparams);

            // compute energy contribution
            double einc = LennardJones(r, epsc, sigmac, params.rLJcutoff *
sigmac);

            if (dist == params.neighbor_ignoredist)
                einc *= params.fudgeLJ; // scale LJ-potential for (typically)
1-4 interactions

            ljenergy += einc;
        }
    }

    // compute LJ long-range correction (this should be pre-computed,
just scaled by volume)
    double box2volume = state.crystalparams[0] * state.crystalparams[1] *
state.crystalparams[2];
    for (unsigned int atidx1 = 0; atidx1 < atomtypes.size(); atidx1++) {
        for (unsigned int atidx2 = atidx1+1; atidx2 < atomtypes.size();
atidx2++) {
            sigmac = (atomtypes[atidx1].sigma + atomtypes[atidx2].sigma) /
2.0;
            epsc = sqrt(atomtypes[atidx2].epsilon *
atomtypes[atidx2].epsilon);

            ljenergy += LennardJonesCorr(params.rLJcutoff * sigmac, epsc,
sigmac,
                moldata.NrAtomsOfType(atidx1) *
moldata.NrAtomsOfType(atidx2) / box2volume);
        }
    }
}

```

```

    return ljenergy;
}

// compute LJ potential between molecules mol1 and mol2
double EnergyComputer::MolMolLJEnergy(const MoleculeData &mol1data,
const MoleculeData &mol2data, const ModelParams &params, const
ModelState &state, const std::vector<AtomTypeData> &atomtypes) const
{
    double ljenergy = 0.0;
    double sigmac, epsc, r[3];
    vector<ModelAtom>::const_iterator ait1, ait2;

    // compute LJ-contribution
    for (ait1 = mol1data.atoms.begin(); ait1 != mol1data.atoms.end();
ait1++) {
        for (ait2 = mol2data.atoms.begin(); ait2 != mol2data.atoms.end();
ait2++) {
            // combination rules for sigma, eps
            sigmac = (atomtypes[ait1->idx].sigma + atomtypes[ait2-
>idx].sigma) / 2.0;
            epsc = sqrt(atomtypes[ait1->idx].epsilon
* atomtypes[ait2->idx].epsilon);

            // compute distance and map periodically
            vdiff(ait1->x, ait2->x, r);
            mapdist_to_nearest(r, state.crystalparams);

            ljenergy += LennardJones(r, epsc, sigmac, params.rLJcutoff *
sigmac);
        }
    }

    // compute LJ-correction (should be precomputed)
    double box2volume = state.crystalparams[0] * state.crystalparams[1] *
state.crystalparams[2];
    for (unsigned int atidx1 = 0; atidx1 < atomtypes.size(); atidx1++) {
        for (unsigned int atidx2 = atidx1+1; atidx2 < atomtypes.size();
atidx2++) {
            sigmac = (atomtypes[atidx1].sigma + atomtypes[atidx2].sigma) /
2.0;
            epsc = sqrt(atomtypes[atidx2].epsilon *
atomtypes[atidx1].epsilon);

            double rho = (mol1data.NrAtomsOfType(atidx1) +
mol2data.NrAtomsOfType(atidx1))
                * (mol1data.NrAtomsOfType(atidx2) +
mol2data.NrAtomsOfType(atidx2)) / box2volume;

            ljenergy += LennardJonesCorr(params.rLJcutoff * sigmac, epsc,
sigmac, rho);
        }
    }

    return ljenergy;
}

```

```

// computes the internal real part of the Ewald summation for one
molecule
double EnergyComputer::InternalEwaldRealEnergy(const MoleculeData
&moldata, const ModelParams &params, const ModelState &state) const
{
    double erenergy = 0.0;
    vector<ModelAtom>::const_iterator ait1, ait2;
    double sqrtalpha = sqrt(params.alphaEw);
    double einc, r[3];

    for (ait1 = moldata.atoms.begin(); ait1 != moldata.atoms.end();
ait1++) {
        for (ait2 = ait1+1; ait2 != moldata.atoms.end(); ait2++) {
            int dist = moldata.NeighborDistance(ait1->idx, ait2->idx);
            if (dist < params.neighbor_ignoredist) // ignore nearest
neighbors
                continue;

            vdiff(ait1->x, ait2->x, r);
            mapdist_to_nearest(r, state.crystalparams);

            einc = EwaldReal(ait1->charge, ait2->charge, sqrtalpha,
sqrt(dot(r,r)), params.rEwcutoff);

            if (dist == params.neighbor_ignoredist)
                einc *= params.fudgeQQ; // scale LJ-potential for (typically)
1-4 interactions

            erenergy += einc;
        }
    }

    return erenergy;
}

// computes the real part of the Ewald summation between two molecules
double EnergyComputer::MolMolEwaldRealEnergy(const MoleculeData
&mol1data, const MoleculeData &mol2data, const ModelParams &params,
const ModelState &state) const
{
    double erenergy = 0.0;
    vector<ModelAtom>::const_iterator ait1, ait2;
    double sqrtalpha = sqrt(params.alphaEw);
    double r[3];

    for (ait1 = mol1data.atoms.begin(); ait1 != mol1data.atoms.end();
ait1++) {
        for (ait2 = mol2data.atoms.begin(); ait2 != mol2data.atoms.end();
ait2++) {
            vdiff(ait1->x, ait2->x, r);
            mapdist_to_nearest(r, state.crystalparams);

            erenergy += EwaldReal(ait1->charge, ait2->charge, sqrtalpha,
sqrt(dot(r,r)), params.rEwcutoff);

```

```

    }
}

return ereenergy;
}

double EnergyComputer::ComputeInternalEnergy(const MoleculeData
&moldata,
    const ModelParams &params, const ModelState &state, const
vector<AtomTypeData> &atomtypes)
{
    double totenergy = 0.0;
    double bondenergy = 0.0, angleenergy = 0.0, torsionenergy = 0.0,
ooplaneenergy = 0.0;

    vector<AABond>::const_iterator bit;
    for (bit = moldata.bonds.begin(); bit != moldata.bonds.end(); bit++)
        bondenergy += BondEnergy(*bit, moldata.atoms);

    vector<AAAngle>::const_iterator ait;
    for (ait = moldata.angles.begin(); ait != moldata.angles.end();
ait++)
        angleenergy += AngleEnergy(*ait, moldata.atoms);

    m_dih1E = 0.0; m_dih3E = 0.0;
    vector<AADihedral>::const_iterator dit;
    for (dit = moldata.propers.begin(); dit != moldata.propers.end();
dit++)
        torsionenergy += DihedralEnergy(*dit, moldata.atoms);

    for (dit = moldata.impropers.begin(); dit != moldata.impropers.end();
dit++)
        ooplaneenergy += DihedralEnergy(*dit, moldata.atoms);

    totenergy = bondenergy + angleenergy + torsionenergy + ooplaneenergy;

    cout << "Bonds: " << bondenergy << ", Angles: " << angleenergy
        << ", Torsion: " << torsionenergy << ", Out-of-plane: " <<
ooplaneenergy << endl;

    return totenergy;
}

```

Main.cpp

```

/*
 * File:    main.cpp
 * Author:  TOGE
 *
 * Created on den 16 september 2009, 16:02
 */

```

```

#include "FFloader.h"
#include "EnergyComputer.h"
#include "FortranInterface.h"
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <string>
#include <cstring>

using namespace std;

char default_datafile[] = "data.dat";

/*
 *
 */
int main(int argc, char** argv) {
    FFloader loader;
    EnergyComputer eman;
    ModelData mdata;
    string datafilename, fffile, startupfile;
    int startiternr;
    bool loadMM = true;

    datafilename = default_datafile;

    if (argc >= 2) {
        for (int k=1; k<argc; k++) {
            if (argv[k][0] == '-') {
                // parse switches
                if (strcmp(argv[k]+1,"noMM") == 0)
                    loadMM = false; // no
macromolecules
            }
            else {
                datafilename = argv[k];
            }
        }
    }

    if (loadMM) {
        if (!ReadFileNames(datafilename, fffile, startupfile,
startiternr)) {
            cerr << "Error reading data file: " << datafilename
<< endl;
            return 7;
        }

        loader.SetDebugMsgLevel(2); // set level for printing
messages, 2 is normal

        if (loader.Load(fffiler.c_str()) != 0) {
            cerr << "Could not open force field files! Stem: "
<< fffile << "\n" << endl;
            return 1;
        }
    }
}

```

```

DecorateFileNameCPP(startupfile, "2mm", startiternr);
if (loader.LoadSystemFromPDB(startupfile.c_str(), mdata) !=
0) {
    cerr << "Could not open pdb-file " << startupfile
<< "\n" << endl;
    return 2;
}

cout << "# chains loaded: " << mdata.chains.size() << endl;
for( int chainnr=0; chainnr < mdata.chains.size() ;
chainnr++ ) {
    cout << "chain " << chainnr+1 << " #
molecules loaded: " << mdata.chains[chainnr].molecules.size() << endl;
    for (int molnr=0;
molnr<mdata.chains[chainnr].molecules.size(); molnr++) {
        double charge = 0.0;
        for (int atnr=0; atnr <
mdata.chains[chainnr].molecules[molnr]->atoms.size(); atnr++)
            charge +=
mdata.chains[chainnr].molecules[molnr]->atoms[atnr].charge;
        cout << "Charge of molecule " << molnr
<< ": " << setprecision(3) << charge << endl;
    }

    int molnr = 0;
    cout << "Molecule " << molnr+1 << ": "
<< mdata.chains[chainnr].molecules[molnr]->atoms.size() << " atoms, "
<<
mdata.chains[chainnr].molecules[molnr]->bonds.size() << " bonds, "
<<
mdata.chains[chainnr].molecules[molnr]->angles.size() << " angles, "
<<
mdata.chains[chainnr].molecules[molnr]->propers.size() << " propers, "
<<
mdata.chains[chainnr].molecules[molnr]->impropers.size() << "
impropers." << endl;

    cout << "Total internal energy
is " <<
eman.ComputeInternalEnergy(*mdata.chains[chainnr].molecules[molnr],
mdata.parameters, mdata.currentstate, mdata.atomtypes) << endl;
}

}
else {
    cout << "Running simulation without macromolecules!\n" <<
endl;

    // We next call RunFortranSimulation with mdata.size() == 0,
i.e. Nmmol == 0
    // which is ok: loads or creates small molecule config and
runs sim with small mols only
}

RunFortanSimulation(datafilename.c_str(), mdata);

```

```
    return 0;  
}
```