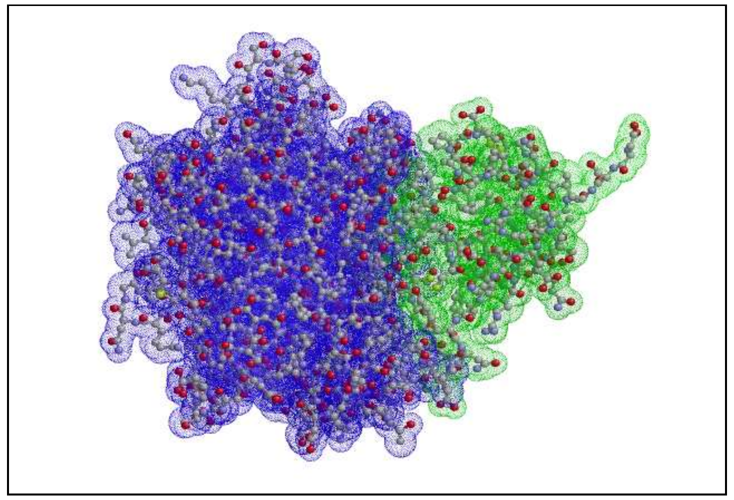
Protein–Protein interaction; Heuristic search in databases of 3d protein structures



Master thesis of
Hans–Henrik Stærfeldt
Datalogisk Institut
Københavns Universitet

Marts, 2001

Protein–Protein interaction; Heuristic search in databases of 3d protein structures

Hans–Henrik Stærfeldt

October 2, 2002

Abstract

In protein research and drug development and when working with new proteins it is often important to determine how and with which other proteins interaction occurs. Similar it is of interest to know how these complexes are arranged in 3 dimensions.

Such detailed knowledge can help define any biological role such as inhibition or enhancement of different functions in the biochemical pathways, as well as help find drugs that will affect the compounds. One goal of this work is to implement a heuristic search technique which is able to search against a library of 3d protein structures given a query proteins 3d structure. The implementation seeks out promising candidates to which the query protein is likely to bind, as well as an approximate configuration of the resulting compound. This is done using a shape complementarity measure, a simple electrostatic interaction model based on partial charges of the protein atoms, and a desolvent measure.

The search uses a technique of sampling the energy fields and density of the proteins in a 3d grid allowing for fast ranking of translational configurations to be reached by Fourier correlation theory, as well as speed up the search by clustering the rotational space and applying a branch–and–bound algorithm for the search. The method takes into account flexible amino acids by constructing an energy measure which is a bound for the average interaction energy between all possible configurations of the amino acids. However backbone flexibility is not considered explicitly.

The method implementation was not finished at the time this report was due for evaluation. This work therefore describes what methods have been used as solutions to some of the partial problems, as well as which design features were planned.

Contents

Chapter 1

Acknowledgments

Much work has gone into this project, and i would like to thank people for helping me and supporting me during the time it was produced. I would like to thank Lars Juhl Jensen for always wanting to discuss the various subjects of molecular biology and physical chemistry at which i were not an expert. My girlfriend Maj-Britt Kryger has been very supporting during my all-nighters of programming sessions that went into the work, as well as help me get the most gruelling examples of bad formulation ironed out. I would like to thank my thesis advisor Jyrki J. Katajainen for supportive and encouraging advice during the project.

Chapter 2

Introduction

Characteristically for protein interactions is that the proteins must bind to each other in order to perform their biological roles. Either in transportation, catalysis, inhibition, or to form complexes to carry on a role e.g. enzyme cooperativity.

The work on protein association has over the years evolved from the experimental study of multimeric proteins to also incorporate computer analysis of protein–protein interaction models.

This branch of study had not been possible, if not for the continued collection of protein structures resolved through X-ray crystallography and nuclear magnetic resonance. The worldwide collection of this data in the Brookhaven Protein Databank [4] has allowed researchers access to many structures for analytical comparison and analysis.

A very important aspect of protein interaction is protein symmetry, in that protein structures are able to arrange themselves into arrays of the same protein with a high degree of symmetry.

In light of this a very popular problem is to computationally be able to reliably predict which molecules or proteins a given protein may bind to, and what configuration they will assume in the bound state. Such a configuration includes in principle determination of all the degrees of freedom that a protein may take, being primarily the 6 degrees of rotational and translational freedom, but also the many thousands degrees of freedom inherent to the internal configuration of the proteins and the binding molecules.

Often the methods described in literature focus either on the fine tuning of the internal degrees of freedom in a known 6d configuration, and most often for smaller chemical molecules which bind to the protein. This focus is due to the fact that is is very useful in drug design, and can be used to design drugs that interact with specific known functional sites in proteins, thus computationally locating which drugs should be tested out of a large database of possible chemical compounds. If specific enough, a computational method could also serve to screen a given drug from undesired interaction with other compounds than the one targeted for the drug design.

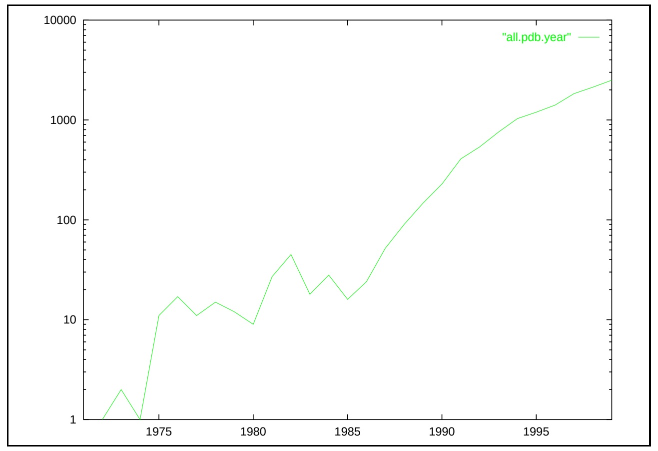


Figure 2.1: The growth of entries in the Brookhaven Protein Databank (PDB)

Recently other work has been targeted at the much more complex problem of protein–protein interaction. The problem is hard because the proteins may have many configurational changes when they bind, and finding optimal binding configurations potentially requires resolving many degrees of freedom.

In this problem, focus is often on determining the 6d configuration of the binding from rigid structures without taking much note of the internal structural freedom of the proteins, and even though the methods uses some heuristic evaluation functions to score good configurations high, their success is often limited by the requirement that the internal configurational change in the proteins is relatively small when they bind to one another.

2.1 Problem description

This works seeks to design and develop a method that is to be used to search for possible binding configurations (translation and rotation) between proteins, taking into account some of the flexibility of the proteins. Computationally the proteins are given as the resolved 3d protein structures in Brookhaven Protein Database (PDB) [4] format, and optimally a match would describe a protein binding configuration that would occur in the cell. Figure 2.1 shows the almost exponential growth of this database over the years.

The search would be made against a database of 3d protein structures to find any possible matches to a query 3d protein structure, where a match constitute an approximate configuration that is likely to occur between the query 3d protein structure and the target elements in the database. This would allow researchers to asses functional dependence between different proteins, and with this knowledge optimize their experiments and save valuable resources in discovering a query proteins relationship with other proteins in the cell.

As the proteins may change structurally depending on the situation, and since neither the 3d structures of the proteins, which are searched against, nor the the search protein may be sampled in the 3d form in which they would natively bind to the each other, the search need to take into account some degree of the changes that would occur.

Finally there is the problem of choosing appropriate energy measures to evaluate a given configuration of the proteins interaction.

Part of this work has been to get an overview of the factors that are found to be important in protein association, to be able to design and implement the heuristic search functions needed to rank the possible configurations configurations.

2.2 Proteins

The proteins have both essential structural and functional roles in the cell. Proteins are therefore important in the study of biology and biochemistry. The protein is a sequence of *amino acids* [1]. These amino acids are linked together through peptide bonds so that they form a chain, where the *backbone* of the chain is similar for each amino acid, but the residual group (called *side-chain* or *R–group*) varies between 20 different amino acids. The functionality of the protein is determined by the 3d structure it assumes in the cell. The segments of the polypeptide backbone of the protein allows it to change into the configuration in which it is functional (also known as *folding*).

Figure 2.2 shows a segment of the backbone, and its degrees of freedom that allows the backbone to change shape. The degrees of freedom is often referred to a *phi/psi–angles* describing the dihedral angles of the carbon–carbon and carbon–nitrogen bonds in the chain. The peptide links are held in a plane as shown, because of a resonance of a covalent double bond shared by nitrogen and oxygen over a carbon atom.

The phi angle is defined as the dihedral angle between the atoms for a given amino acid (i) backbone atoms $C_{i-1} - N_i - C\alpha_i - C_i$ while the psi angle is defined as the dihedral angle between the atoms $N_i - C\alpha_i - C_i - N_{i+1}$.

2.2.1 Structure

The structure of the proteins are often considered in 4 terms; *Primary*, *Secondary*, *Tertiary* and *Quaternary*. The primary structure of the protein is simply the sequence of its amino acids. This structure is the most common, as it can often be inferred by the organisms genomic sequence, or otherwise determined experimentally. The secondary structure of the protein relates to local 3d configuration that is observed along the protein backbone. These configurations have

[1] Except proline which is an *imino acid*

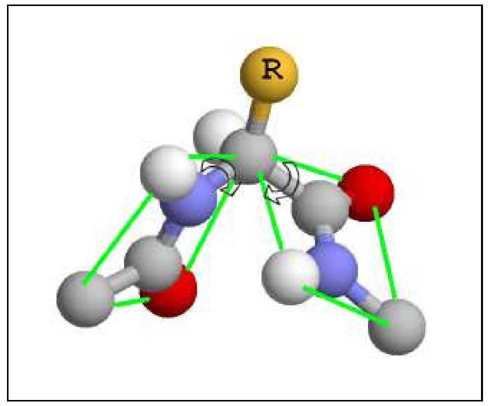


Figure 2.2: The polypeptide backbone, ands its degrees of freedom. The peptide chain links are planar (green) due to a shared covalent double bond. The carbon atoms are grey, the oxygen atoms are red, and the nitrogen atoms are blue.

been classified into a number of different classes, but most notable are *helices* and *sheets*. In Figure 2.3 (left) this local structure is illustrated in an emphasized way, where the the helix is marked red, and the sheet is marked as yellow. Deriving the secondary structure computationally from the primary structure [1] has been fairly successful. The tertiary structure, which is the 3d coordinates of the atoms of the protein, require laboratory work in isolating and refining the protein and measuring its structure using NMR [2] or X-ray crystallography. In Figure 2.3 (right) an example of such a full structure is shown. Many attempts have been made to derive this structure directly from the primary structure, but so far success has been limited. One limitation to a computational solution to the problem is that the backbone and the side chains have many degrees of movement possible, making exhaustive optimization hard. The quaternary structure refers to the conglomeration of several protein into protein complexes. This structure can also sometimes be derived with X-ray crystallography and NMR analysis of the compound, but the increased complexity of the compound proteins makes it technically difficult to determine large complexes. The proteins in a quaternary structure may function as *receptors* and the molecules that bind to them are referred to as *ligands*. Usually the word ligand refers to a smaller non–protein compound, but I will use it forthwith to also mean the smaller protein element that changes the function of the receptor by binding with it. The area in which the receptor and ligand reacts are referred to at the *interface* or *binding site* between them. The interface atoms has been shown to pack very closely together [10].

When resolving tertiary protein structure of a receptor and a ligand that is

[2] Nuclear Magnetic Resonance

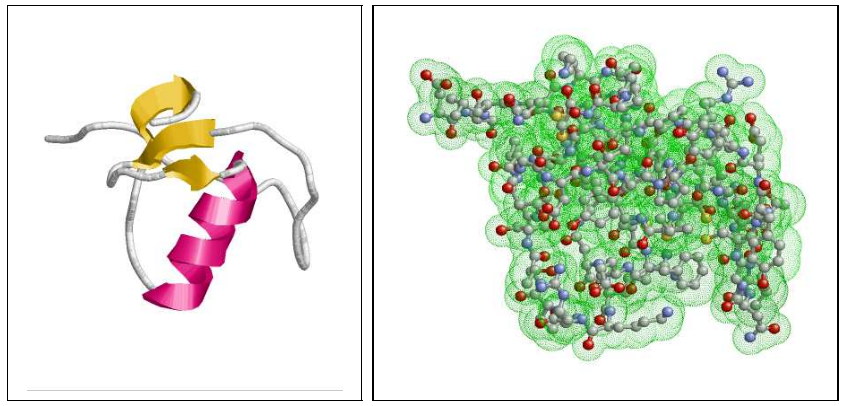


Figure 2.3: Backbone secondary structure (left) and side-chains and surface (right) of ligand human pancreatic trypsin inhibitor variant 3

bound to each other, the structures are said to be *bound* with respect to that particular configuration. If the structure is determined independently of such an interaction, the structure is said to be *unbound*. Determining how a receptor and a ligand bind to each other is also know as *docking*.

2.2.2 Degrees of freedom

Besides the degrees of freedom inferred by the rotations of the backbone of the protein, the individual amino acids attached to the backbone may also have several degrees of freedom. They may rotate around their single atom bonds to create many different configurations. These degrees of freedom are particularly important in protein–protein interaction, as the side-chains change configuration to fit a particular interaction between a receptor and a ligand. The configurations of the side-chains are often described as *chi–angles* listing the dihedral angles formed by the rotations around the side-chains single atom bonds. Several chi angles is needed to describe the more complex side-chains (such as arginine shown in Figure 2.4). In designing a program this change is more manageable, as the consequence has a reasonable local effect limited by the size of the side-chain.

A search method need to take into account some of these degrees of freedom when evaluating energy functions for a given configuration between two proteins, if it wants to be able to evaluate the energy between unbound 3d protein structures.

However the backbone may also change as a receptor and ligand interact. This change is however much worse than the local effects of the side-chains, as a rotation effects all the atoms each side of the rotation, making the charges decidedly global in nature. However, many interactions have been found to change the backbone structure only a small amount when bound.

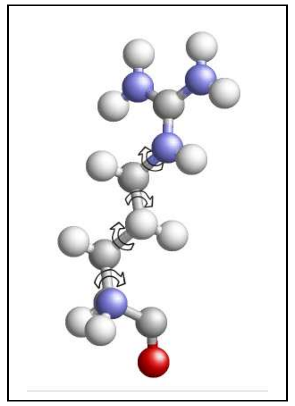


Figure 2.4: The arginine side-chain, and its degrees of freedom.

Therefore many attempts to determine protein–protein interaction focuses on the case where the backbones degrees of freedom is either extremely limited to areas of greater flexibility, such as those parts of the backbone that is not involved in secondary structures which make it rigid, such as helices and sheets. Others simply does not consider the backbone changes at all.

2.2.3 Surface area

Often in a receptor/ligand interface only the surface atoms of the proteins plays a direct role in the interaction. Therefore it is helpful to determine which atoms of the receptor and ligand have access to the surface when devising the energy functions used to evaluate a particular configuration. There are several ways to determine the surface of a protein. Traditionally the surface is defined as a property of each atom, that is calculated as the surface spanned by the center of a sphere rolling over the exclusion radius (*van der Waals radius*) of the atoms, also called *Conolly surfaces* [9]. The spheres radius is usually around 1.5 Å[3] approximating that of a water molecule. Another definition is to make a surface based solely on the van der Waals radius of the atoms. In Figure 2.5 the surface of a protein is shown.

The surface is also important when determining the role of the solvent, as surface atoms interact in various ways with the solvent.

2.2.4 Post translational modifications

Apart from the common 20 amino acids which typically make up the building blocks of the protein, there is an increasing interest in post translational modifications of proteins and their role in the proteins functionality.

[3] One Å (Ångstrøm) equals 10^{-10}meters

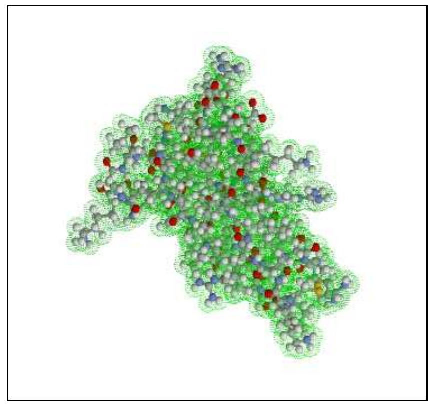


Figure 2.5: The accessible surface area (green) of a protein based on the van der Waals radius of the atoms

In this process side-chains can get modified in various ways, such as sugar attachment, acetylation, cross link with other side-chains or otherwise altering of the protein, as described in [14]. Such a modifications can have significant effect for the proteins ability to take part in specific complexes, especially if the modification is in or near a region of the interface.

Usually post translational modification is not considered unless specifically expressed in the 3d protein structure.

2.3 Protein–Protein interfaces

Many factors have been contributed to the docking problem, and are known to have influence on the 3d structure of the complex of the bound proteins. Part of the problem of finding good docking algorithms is to develop good energy models that favor the natural configurations of the receptor and ligand. In this section i will give a short overview of some of the most commonly described factors.

2.3.1 Electrostatic interactions

Because some of the covalent bonds of the protein are between atoms of different electro–negativity, some of these atoms become polarized by electron delocalization, as one atom donates an electron to fill the other atoms missing electron slots. The *polar* amino acids are Serine, Threonine, Cysteine, Methionine, Asparagine and glutamine. As atoms form electric dipoles by this delocalization, some parts of the side-chains will appear to carry small charges. These charges may act with coulombic force in the protein–protein interface to either attract or repel charges

```
R--H ··· D
```

Figure 2.6: The hydrogen bond is formed if the hydrogen (H) is covalently bonded with a electro–negative receptor atom (R), and a negatively charged donor atoms (D) is near it, preferably as close to the covalent bond direction.

on the other protein thus acting to strengthen or weaken the interaction energy between the two proteins at a given configuration.

Some of the atoms may carry a charge under physiological conditions (i.e. PH 7.0), as they interact with the solvent. They vary from the negatively *charged* acidic (Aspartate and Glutamate) to the positively charged basic side-chains (Histidine, Arginine and Lysine). The assignment of charge is basically a function of the solutions PH value, but usually charge assignments based on the cells average PH is done.

The electrostatic force of a configuration is typically approximated with the sum of the force among a set of point charges q_a and q_b (see Equation 2.1). Usually a heuristic measure of the medium electrostatic permittivity is considered rather than the permittivity of vacuum ϵ_0. The point charges are usually centered on the individual atoms, such as in the AMBER forcefield [11].

$$F = \sum_{a \neq b} \frac{q_a q_b}{4\pi\epsilon_0 r_{ab}} \tag{2.1}$$

Here r_{ab} is the distance between the point charges a and b, with the charges q_a and q_b.

2.3.2 Hydrogen bonds

A specific and very common example of polar interaction is when the hydrogen atom forms a *hydrogen bond* with nearby atoms (see Figure 2.6). These bonds occur when the hydrogen atom is electro–positive in a covalent binding with a more electro–negative atom, such as oxygen and nitrogen, and may form weak bonds with electro–negative atoms.

These bonds have been found to play a crucial role in protein–protein interaction as well as the formation of secondary structure [1] and have been previously used in protein docking programs [18, 19]. The hydrogen bond dipole field is highly directional in its strength, bonds that lie in a direct line between the donor atom, the hydrogen atom, and the acceptor atom are the strongest.

2.3.3 Shape complementarity

One way of classifying the potential of a configuration between the receptor and the ligand is to assume that the area in which they bind will have shape complementarity. Studies of this [22] confirm that shape complementarity plays an important role in 'recognizing' protein–protein interfaces, even when tried with unbound structures [7].

Shape complementarity has been used to make some measures of compatibility [15], but has also been found to perform better when other energies such as hydrogen bonds [18], electrical effects and desolvation effects [7] are taken into account. The surface complementarity has been found to behave particularly well for bound protein structures, but has also been partially successful in identifying unbound interaction.

This suggests that overall shape in some cases does not change much when the proteins bind.

2.3.4 Hydrophobicity

The term *hydrophobicity* refers to an amino acids ability to be insoluble in water. As different amino acids have varying polarity and charge, and thus some does not get hydrated, they are not soluble, or almost insoluble in water. Because of this, they are more often found in the core of a protein, where they are not in contact with the surrounding water environment.

It has been shown that hydrophobicity also plays a role in the interaction between proteins [32, 26], as the protein–protein interface that contain hydrophobic side-chains may cause these to be shielded from the water when two proteins interact, thus reducing the overall energy of the protein by increasing its solubility in water. Statistically this shows as a slight tendency to have charged (*hydrophilic*) side-chains as part of a protein–protein interface, however separate interface classes have been shown to have significantly higher propensity of polar while others have been shown to have higher propensity of non–polar residues [10].

2.3.5 π orbital interactions

The electrons from the covalent bonds on aromatic rings (such as tyrosine) form π – orbitals that electrically leaves the aromatic ring as a positive plane of atoms sandwiched between two negatively charged rings of delocalized electrons above and below it (see Figure 2.7. It has been shown that the strong electron delocalization that is created in aromatic rings have effects on the binding process such as reported by Pletneva et al. [25]. Other π – orbitals in the protein molecule can also have effect on the binding, as described by Andersen [1], and could help determine interaction energies. If the charge model applied in the energy eval-

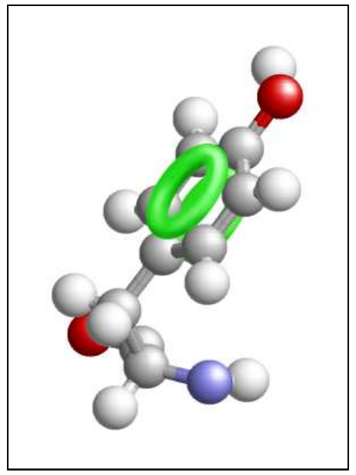


Figure 2.7: Illustration of the shape of the orbitals (green) in the aromatic part of tyrosine. The negative charge distribution electrons is distinct from the atom localization, making atom–based charge models unable to model this.

uation is localized to the atoms e.g. the AMBER forcefield [11], these electrical effects are not considered. As aromatic ring interaction is very complex, they are usually not considered in protein docking, but only in molecular dynamics approaches.

An analysis of protein interfaces by Conte et al [10] also reveals that there exist a much higher propensity of aromatic residues in the protein–protein interface than in the average (21% vs 8%) which implies that aromatic residues have an active functional role in the interface.

2.3.6 Van der Waals interaction

Finally there is the contact attraction and repulsion energies related to atoms. These energies are most important in interactions where the atoms are uncharged, as these energies are often insignificant compared to the interaction energies of charged atoms. The Van der Waals interaction occur due to the atoms core and their electron orbitals create a weak dipole which works to attract other atoms by polarization. The force is usually presented as an attractive and a repulsive force. The repulsive force is not well understood, but often approximated, such as in the Lennard–Jones potential (see Equation 2.2 and Figure 2.8);

$$E = \frac{\epsilon(r_{\mathrm{vdwa}} + r_{\mathrm{vdwb}})^{12}}{r_{ab}^{12}} - \frac{2\epsilon(r_{\mathrm{vdwa}} + r_{\mathrm{vdwb}})^{6}}{r_{ab}^{6}} \tag{2.2}$$

Lennard-Jones 12-6 potential for van der Waals interaction between 2 atoms a and b. The r_{ab} is the distance between a and b, and r_{vdwa} and r_{vdwb} is the van

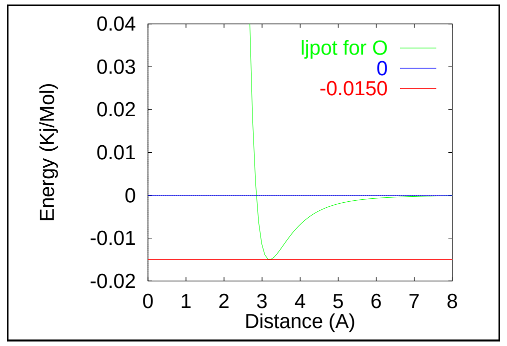


Figure 2.8: The Lennard–Jones 12–6 potential as a function of distance between atoms.

der Waals radii of a and b respectively

2.4 Protein docking

The main goal of protein docking is to determine whether two proteins have an energetically favorable binding configuration. This knowledge can be used by drug designers to develop drugs that inhibit or in other ways change the function of a given protein. Also, the exact configuration of such a favorable interaction is desired, as this can lead to knowledge of what exact effects the interactions may have.

Usually the methods that are implemented assume knowledge of which region on the proteins constitute the interface, and most of the methods have been focused on the important task of docking smaller chemicals into these regions for the purpose of drug design, as for example in the GREEN package [31], and many others.

The success of protein docking methods are partly dependent on choosing an energy function that correctly identifies the configurations that occur in the cell. Several aspects of the chemical reactions has been considered in this problem, but most prominent is shape complementarity and hydrogen bonds.

A secondly difficulty is that the interaction may change the configuration of both proteins among their extreme multitude of degrees of freedom, which excludes an exhaustive search of the configuration space. In the case where the ligand is not a protein, but a smaller molecule, the number of degrees of freedom becomes more manageable, but the methods does not extend into full protein–protein interaction. One of the problems, when conducting a full search of the degrees of freedom, is to identify which configurations are sterically viable and thus eliminate the search space by excluding unfavorable or impossible parts of it, as for example where atomic cores overlap.

Therefore solutions to protein–protein docking often make various shortcuts to the problem, limiting the search space en various ways.

In *static docking*, no atoms are allowed to move relative to each other in the process of searching for an optimum configuration. This method usually work well when trying to dock bound proteins, as the atoms in these structures are positioned 'optimally' for a particular interface.

In *semi flexible docking*, or *soft docking* only partial flexibility is allowed. Usually the backbone of the receptor is fixed during docking, while side-chains are allowed free movement (under the usual chemical limitations). Other solutions may be to derive heuristic energy measures which reduce the penalty of stearic overlap between surface atoms. As the side-chains of the proteins are considerably shorter than the backbone, these configurational searches have spatially limited effects. Some methods also allow limited altering of the backbone configurations, in changes that does not imply long range movement of the proteins, but is limited to sets of backbone conformational search to spatially localized changes. An example of this is loop energy minimization where a section of the backbone with low rigidity is allowed to move, but its endpoints are fixed.

Full *flexible docking* allows for all rotational and translational degrees of freedom of the atoms to be considered. This type of docking is often computationally very demanding, as it relies on full molecular dynamics using Monte–Carlo simulations to incrementally determine interactions. It is therefore unsuitable for database searches, and often requires previous knowledge of the location of the binding site.

2.4.1 Other work

Most work in this field is focused on the later stage of drug discovery, and are thus designed to dock small non–protein ligands into known receptors of receptor proteins, where the ligand is allowed flexibility, while the receptor often is not. The methods used in previous works are very diverse, but mostly the methods uses shape complementarity to extract a limited number of configurations that is then cored using various methods.

In work such as in Claußen et al [8] the problem of flexible side-chains is considered. This method relies on obtaining samples of varying structural configurations (between 4 and 16 according to the article) within the known docking site, and incorporate these into an optimal solution for a given small, flexible ligand. The method does however not consider protein–protein interaction.

Bacon et al [2] use a shape complementarity measure by superimposing points on the surface of protein and ligand, and optimizing them using a least–squares best–fit algorithm to limit the number of conformation to be ranked by an electrostatic scoring function. The method address only flexibility through brute force sampling of conformations, which in practice excludes protein–protein interaction to be considered. Norell et al [21] also develops a shape complementarity measure defining 'knobs' and 'holes' which are matched geometrically to serve as start points for chemical refinement and energy scoring, limiting the translational and rotation search space enormously. This method does however not consider flexible side-chains.

Meyer et al [18] and Mizutani et al [20] use hydrogen bond estimations between the ligand and the protein to reduce the search space under the notion that hydrogen bonds should be part of the interface. While Mizutani et al only works with small ligand interfaces, Meyer et al has and extended measure that cover protein–protein interfaces, defining a limited number of interaction points only where favorable hydrogen bonds are present. However the methods uses only rigid body protein definitions, and this makes it hard to apply to unbound docking, since the side-chains may be at a energetically unfavorable configuration for the natural interface.

In later work Norell et al [22] further examines the effect of unbounded protein structures on the shape complementarity scheme, using the previously defined 'knobs' and 'holes' method on the receptor and ligand of unbound proteins, and then match them with each other. Their results reveal that using shape complementarity as a filter to limit the search space will even work for some unbound proteins, reporting RMSD results of up to 5 Å.

Work such as Rarey et al [27, 26] also work with various ways to limit the search space, such as selecting only configurations that show favorable hydrogen bonds and salt bridges in the interface. However, their method only extend to a limited number of conformations of a small ligand, and the process is complicated when docking fully flexible, but very short, peptides and requires user intervention. However in this case, even the backbone of the ligand was considered flexible.

In the work of Yang et al [33] ligand flexibility was searched though evolutionary algorithms, where the configurations are represented as genetic information, which is then evaluated using a scoring function. This method uses a new algorithm called 'family competition evolutionary algorithm' and in this particular case it is augmented with a method that adjusts the algorithm parameters on the fly. Using a cross breeding of genetic algorithms with the molecular dynamics package CHARMM [6] Taylor et al [30] had good results docking fully flexible small ligands to a rigid protein structure.

Jiang et al [16] uses grid–based approach for soft surface matching using surface cubes, but the method seems to be applicable to small ligands only. Also Helmer–Citterich et al [15] used shape complementarity in a grid–based approach to obtain favorable results with bound, static protein–protein interaction, but did not consider neither receptor nor ligand flexibility.

Unbound docking was attempted by Chen et al [7] and the flexibility of side-chains were incorporated through heuristic scoring measures. Shape complementarity measure as well as desolvent effects and electrostatics were considered in a method using a cartesian energy field/density grid representations and using

Fourier correlation analysis to search through the translational degrees of freedom. Here the shape complementarity measure were softened by limiting the penalties for stearic overlap of atoms which were found on the surface of the proteins. This method was able to report fairly good results even for the harder of the unbound cases.

Ritchie et al [28] also uses Fourier analysis to evaluate surface complementarity very similar to that of Chen et al, but chooses an elaborate spherical–polar Fourier analysis in contrast to the cartesian one. The method suffers from the need to precalculate overlapping scaling function for the proteins that need to be docked. Also spherical–polar Fourier analysis does not have a fast evaluation algorithm similar to the cartesian Fourier analysis, and rather than a grid–based sampling approach, the method relies on making a symbolic analysis of approximated polynomial. However the method gains in that these functions need only be computed once, and that the correlation of the functions become more speedy, making an actual search against several proteins more viable.

A soft docking approach by Palma et al [24] treats the side-chain flexibility in a full 6D search of the translational and rotational by applying a soft binary surface function in a grid with bit–size cells and making fast evaluations using bitwise AND functions for fast implementation. Explicitly flexibility of side-chains were here considered by allowing stearic overlap between atomic volumes, if the volume belonged to highly flexible amino acids. The surface matching algorithm served as a filter to select highly surface complementary configurations that then were evaluated using a energy based scoring function. The method did not use Fourier correlation theory, but rather a heuristic grid search that excluded testing core overlaps.

Chapter 3

Docking algorithm development

The main goal of this work is to construct a protein-protein docking algorithm which given the resolved, unbound 3d structure of the receptor and ligand is able to suggest near native binding configurations for their interaction. The method should also incorporate a way to search larger databases of protein structures for possible binding configurations. Furthermore the method should directly address the problems of side-chain flexibility of the proteins to docked.

Early on, I decided to explore the possibilities of using Fourier correlation analysis (see Section 3.1.1) to set up bounds for the interaction energies in ways that let me reduce the vast search space of rotational configuration and side-chain flexibility. The reason for the explicit treatment of the side-chain flexibility is that it has been found that protein–protein complexes with large interaction surfaces tend to undergo internal configurational changes when binding [10].

The development of the algorithm was planned to work in several stages. The first stage seeks to identify the important energy function parameters which are needed for a successful protein-protein docking algorithm, and implement the data manipulation which is needed to further develop the docking algorithm.

The first stage implements a soft, but static, docking algorithm and implements the energy functions as function correlations. This is needed to allow for the use of Fourier correlation theory to effectively find optimal translational configurations.

The second stage seeks to subdivide the rotational search space in an intelligent way, both to ensure even sampling of the rotational space, but also to prepare for a hierarchical subdivision of this space in a way that allows for branch–and–bound searches through the rotational subdivisions.

The third stage of the docking algorithm development implements energy functions that can be bounded given several superimposed configurations of the rotational space. These energy bound derivatives then allow for the implementation of a branch–and–bound search to be implemented on the rotational space subdivision.

The fourth stage implements flexibility of the side-chains by implementing sampling of the possible internal configurations of a side-chain dependent rotamer library. The function correlation energy measures are again extended to take into account the side-chain flexibility and to produce bound for the best interaction energies.

Fifth stage implements a full system of protein–protein docking making a full 6 dimensional search of possible binding configurations of two proteins, using energy bounds to limit search space through a branch–and–bound rotational search, and implementing explicit bounds on energy given a large sampling of side-chain flexibility.

Sixth stage explores the possibilities of clustering a database of proteins in a way that can lead to energy bound of an entire cluster of proteins, using the energy bound techniques developed in stage five.

Final step consist of an evaluation of the method both in sensitivity, but also specificity given a set of examples of unbound or half-bound test cases supplied externally [7].

3.1 Grid–based energy evaluation

I have chosen to use a grid–based approach. There are two main reasons for this. First it allows me to employ Fourier correlation theory to search through the possible translations of two grids describing the interaction energies as discreet samples of *density* and *field* functions related to the grid–cells.

Secondly it allows me to superimpose several configurations, and representing several degrees of freedom (both rotational and side chain flexibility) to create bounding functions for interaction energy.

There exist several drawbacks as well for this approach. Grids take up a lot of memory (32 MB per complex-value grid of size $128 \times 128 \times 128$). However with memory being abundant on big supercomputers today (32 Gb memory is not uncommon), I do not see this as a hindrance. Furthermore all energy evaluation must be of the form of function correlations if Fourier correlation analysis is to be applied. As this is not always possible to transfer various energy measures to this form, some measures may have to be approximated. The size of the grid–cells also determine their accuracy. As the number of grid–cells scale with $O(n^3)$, where n is the number of cells in a given direction, decreasing the cell sizes by a given factor $\delta, \delta < 0$ would require $O(\frac{1}{\delta}^3)$ more cells for the grid to cover the same space.

3.1.1 Fourier correlation analysis

The grid–based Fourier correlation method builds on the Fourier correlation theorem. The Fourier correlation theorem states that given two function f and g

then;

Let $f \star g$ denote the cross correlation between f and g

$$f \star g = \int_{-\infty}^{\infty} \overline{f(\tau)} g(\tau + t) d\tau \tag{3.1}$$

So the new function has a value in every point t, that is the integral of the multiplum of all the values of f and g, where g is shifted by t.

Let $\mathcal{F}$ be the Fourier transformation of a function. Let $F(v) \equiv \mathcal{F}(f(t))$ and $G(v) \equiv \mathcal{F}(g(t))$. Then

$$f \star g = \mathcal{F}(\overline{\mathcal{F}(f)}\mathcal{F}(g)). \tag{3.2}$$

More specifically, since the analysis is done on a discreetly sampled grid, it is discreet Fourier transforms that are used. As the grids are in 3 dimensions, 3 dimensional transforms are used;

Let X be a 3 dimensional complex array with the elements $X[j_1, j_2, j_3]$, where j_1, j_2 and j_3 is as well as i_1, i_2 and i_3 are index of the array. Let $\omega_s = e^{2\pi\sqrt{-1}/n_s}$, $s \in \{1, 2, 3\}$, where n_s is the array dimensions. The 3 dimensional discreet Fourier transform calculates;

$$Y[i_1, i_2, i_3] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_3-1} \sum_{j_3=0}^{n_3-1} X[j_1, j_2, j_3] \omega_1^{-i_1 j_1} \omega_2^{-i_2 j_2} \omega_3^{-i_3 j_3}$$

While the 3 dimensional inverse discreet Fourier transform calculates;

$$Y[i_1, i_2, i_3] = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} X[j_1, j_2, j_3] \omega_1^{i_1 j_1} \omega_2^{i_2 j_2} \omega_3^{i_3 j_3}$$

An important note is that a forward transform followed by an inverse transform multiples the array by $\prod_{s=1}^{3} n_s$. The grids used here are typically equally sized in all 3 dimensions with the size N. Each cell represent a sampled part of space that has a chosen length δ, so that the grids extension in space becomes δN in each direction.

A discreet Fourier transformation (DFT) is different in that it does not have infinite bounds. Rather, the data is assumed to be *periodic*. This means that the frequencies reported in the Fourier transformed 3d arrays are set up to reproduce a *periodic extension* of the grid in 3 dimensions along the axis. This is important because all operations within the Fourier transformed data are treated this way. This means that as the correlation function between the grids move from the center of the grid, the correlation function shows the correlation of a grid that is shifted *modulo* the grid size (See Figure 3.1).

This is important because the grids need to be designed, so that the correlation derived from the discreet Fourier analysis of the grids does not suffer from this

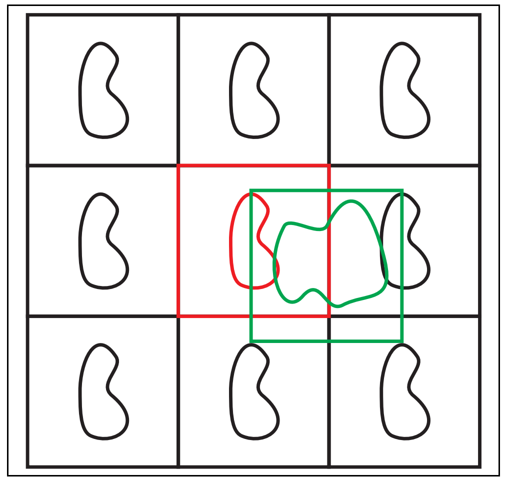


Figure 3.1: The implicit period of DFT. As the correlation function of a given point designates a sum of the multiplum of all points of the field grid A (red) with density grid B (green), and grid B shifted, the actual shift is modulo the size of the grid.

effect. In short, the grids need to be scaled, so that the functions individually does not reach long enough to span the gap between the period.

Applying Equation 3.2 to the discreet case as follows; For two sample grids E_f and E_d we can make a correlation grid $E_f \star E_d$ by applying the discreet Fourier transform DFT and inverse discreet Fourier transform IFT of the data-points;

$$\begin{aligned} E_f \star E_d &= \frac{1}{N^3}\mathrm{IFT}(\overline{\mathrm{DFT}(E_f)}\mathrm{DFT}(E_d)). \\ &= \frac{1}{N^3}\mathrm{IFT}(\mathrm{IFT}(E_f)\mathrm{DFT}(E_d)). \end{aligned} \tag{3.3}$$

The protein is placed in the center of a cartesian grid covering the space around them. By expressing interaction energies in two functions, field and density, in a form such that;

$$E(i,j,k) = \sum_{x,y,z=1}^{N} \mathrm{field}(x,y,z) \times \mathrm{density}(x+i, y+j, z+k) \tag{3.4}$$

And at each cell of that grid, the field and density function is sampled with respect to the current orientation of the protein, the correlation function can then be calculated by applying Equation 3.3.

To ensure that the grid does not suffer from the periodic interaction effect, the grids need to be scaled so that the size of both proteins plus the distance of the longest interaction energy is less than the length of the grid. Only then can discreet Fourier analysis be used to make correlations. This is another problematic side effect of this method, as it makes high-resolution grids even larger.

As the grids are to be reused, it is important that the scaling is correct for all possible rotational angles of the ligand atom. Therefore you must find the smallest sphere that covers all points. As the best exact solutions to the problem involve complex algorithmics (such as vornoi spaces), and the trivial exact solutions are combinatorial in nature, I opted for an approximation, which ensured me a circumscribing circle in $O(n)$ time where n is the number of atoms;

Step 1 Locate the center of gravity (cog) among the atoms. Each atom carrying the same weight.

Step 2 Locate the the atom (a_{max}) that is furthest away from the center of gravity.

Step 3 Circle is centered at (cog) and has radius $|\text{cog} - a_{max}|$

While the resulting sphere is not minimized, except for the specific choice of center, it ensures that enough space is set off to prevent edge effects when using the discreet Fourier transform.

Using the grid–based approach allows for a fast search of possible translations to two field/density grids to be made in $O(n^3 \log(n^3))$ where n is the grid size, by applying the Fourier correlation theory. This general method has been applied before by Chen et al.[7].

3.1.2 Space partitioning tree

To be able to sample the grid–cells effectively, several approaches were attempted to minimize the time needed to set up the correlation functions for the energy measures.

As part of these methods, I have designed and implemented a space partition algorithm, that allows me to query which atoms is within a given radius of a point in space in logarithmic time of number of atoms.

Let $\mathcal{N}$ be a node in such space partitioning tree. Nodes can be of the form *tree-nodes* and *leaf-nodes*. Each node $\mathcal{N}$ covers a 3 dimensional space described by the corners of a box $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$.
Let $\mathcal{N}$.child[1..8] be the 8 children of a tree-node. Let $\mathcal{N}$.atomcount be the total number of atoms in a subtree $\mathcal{N}$. Let $\mathcal{N}$.atoms be a list of atoms registered in a leaf-node $\mathcal{N}$.

```
Insert(LeafNode 𝒩, Atom 𝒜)
    If ( 𝒩.atomcount > log(totalnumberofatoms) )
        SplitCheck(𝒩,𝒜)
    Else
        𝒩.atoms = 𝒩.atoms ∪ 𝒜
    End
Insert(TreeNode 𝒩, Atom 𝒜)
    For (childno = 1..8)
        If ( Covers(𝒜, 𝒩.child[childno]) )
            Insert(𝒩.child[childno],𝒜);
        End
    EndFor
SplitCheck(LeafNode 𝒩, Atom 𝒜)
    Let box[1..8] be the partition of the space covered by 𝒩
    If (all atoms are not covered by the same box)
        Let 𝒩 be a TreeNode.
        Let 𝒩.child[1..8] be its children covering space box[1..8].
        For (childno = 1..8)
            Insert(𝒩.child[childno],𝒜);
        EndFor
    Else
        𝒩.atoms = 𝒩.atoms ∪ 𝒜
    End
Query(TreeNode 𝒩, Position P, Radius R)
    Result = {}
    For (childno = 1..8)
        If ( Intersect(P,R,𝒩.child[childno]) )
            Result = Result ∪ Query(𝒩.child[childno],P,R)
        End
    EndFor
    Return Result
Query(LeafNode 𝒩, Position P, Radius R)
    Result = {}
    For (atomno = 1..𝒩.atomcount)
        If ( Intersect(P,R,𝒩.atoms[atomno]) )
            Result = Result ∪ Query(𝒩.child[childno],P,R)
        End
    EndFor
    Return Result
```

```
Poll(TreeNode 𝒩,Position P_min, Position P_max, Radius R)
    For (atomno = 1..𝒩.atomcount)
        If ( Intersect(P_min,P_max,R,𝒩.atoms[atomno]) )
        If ( Poll(𝒩.child[childno],P_min,P_max,R) == True )
            Return True
        End
    End
    Return False
Poll(LeafNode 𝒩,Position P_min, Position P_max, Radius R)
    For (atomno = 1..𝒩.atomcount)
        If ( Covers(𝒩.atoms[atomno],P_min,P_max,R))
            Return True
        End
    EndFor
    Return False
```

Where *Intersect()* determines if a point and a sphere intersect the space covered by a node, and *Covers()* function determine if an atoms is within the space covered by a node or by another box, extended by a radius in every direction. The implementation also implements removal of atoms, and subsequent merger of nodes, but that functionality is not used in this work.

3.1.3 Grid–cell sampling method

The sampling of the field and density grids of the receptor and ligand in the grid–cells were tried using various methods.

Some of the density and field functions have potentially infinite reach in space (such as the electrostatic interactions). The heuristic solution to this problem is to introduce a maximal distance r_{max} of each interaction, and not consider the sampling of the density and field functions beyond this distance.

The first attempt at a sampling algorithm made a Query call for each grid–cell position, and the merged the values for each of the atoms within that interaction sphere.

As the grid–based method requires that much of the grid needs to be free from the density and field functions to accurately be able to correlate the functions without the risk of edge effects (see Section 3.1.1), many of these regions in the grid will not have any atoms near enough to need their density and field functions to be sampled.

The second attempt subdivides the grid the same way that the space partition tree does (see Section 3.1.2), and then recursively tries to sample the grid–cells of such a subdivision. Before recursively sampling a subdivision, a Poll call is made to determine if any atoms indeed does effect the region at all. This method drastically cuts down the grid–cells that need to be sampled.

The last and most effective method of sampling the density and field functions in the grid–cells was simply to sample the surrounding grid cells of each atom at a time, and use the grid–cells as temporary storage for the accumulated value at a given time. All grid–cells at a distance r_{max} or less from a given atoms center were visited and the density or grid function was accumulated for each atom in this way.

3.2 Choice of energy function

An important part of the work is the choice of energy function. I need to take into account the most important of the interactions of the protein interface. As has been shown by previous work, shape complementarity is often a useful measure, even in the unbound case, if the receptor and ligand do not undergo large configurational changes during binding.

By choosing the electrostatic energy measure, i hope to cover the electrostatics of the interface, including those related to hydrogen bonds.

Some of the energy measures introduced in 2.3 have been left out of the method. Particularly that of the van der Waals interactions. This measure is dificult because it have extremely steep penalties for atomic overlap, and thus is not very soft in its definition. The energy measures chosen here are the ones i have found most frequently used in other work, or covers some of the same functionality.

3.2.1 Surface complementarity

In this work, I have chosen to adapt the surface complementarity functions of Chen et al [7] (See Equation 3.5). The function of Chen et al is closely linked with representing complex numbers.

$$
\begin{aligned}
R_{SC}(c) &= \begin{cases} 1 & \text{c is surface of R} \\ \rho i & \text{c is in core} \\ 0 & \text{c is not near R} \end{cases} \\
L_{SC}(c) &= \begin{cases} 1 & \text{c is in a surface atom of L} \\ \rho i & \text{c is in core} \\ 0 & \text{c is not near L} \end{cases}
\end{aligned}
\tag{3.5}
$$

Where c is a given grid–cell coordinate (i, j, k) in the grid. These surface complementarity density and field functions makes use of complex numbers ($\rho i =$

$\rho\sqrt{-1}$), and ρ is some positive factor (here the value of 8 is used similar to Cehn et al). Using Equation 3.5 the surface measure is then defined by as;

$$\begin{aligned} E_{SC}(i,j,k) &= \mathrm{Re}\Big[\sum_{l,m,n=1}^{N} R_{SC}(l,m,n)\times L_{SC}(l+i,m+j,n+k)\Big] - \\ & \mathrm{Im}\Big[\sum_{l,m,n=1}^{N} R_{SC}(l,m,n)\times L_{SC}(l+i,m+j,n+k)\Big] \end{aligned} \quad (3.6)$$

Where N is the size of the grid, Re means the real part, and Im means the imaginary part of complex number. Using Equation 3.3 with Equation 3.6 the shape complementarity correlation function E_{SC} can be calculated for all grid–cells using discreet Fourier transformations of the grid, after sampling the density and field functions R_{SC} and L_{SC} in two grids.

$$\begin{aligned} E_{tmp} &= \frac{1}{N^3}\mathrm{IFT}(\mathrm{IFT}(R_{SC})\mathrm{DFT}(L_{SC})). \\ E_{SC} &= Re[E_{tmp}] - Im[E_{tmp}]. \end{aligned} \quad (3.7)$$

Chen et al then defines the protein surface for both receptor and ligand using a rolling sphere of 1.4 Å (as described in Section 2.2.3 as an approximation of a water molecule). The core of the receptor is any grid–cell closer than $\sqrt{1.5}$ vdW radius of a core atom, or closer than $\sqrt{0.8}$ vdW radius of a surface atom. The surface grid–cells of the receptor is defined as any grid–cell that is 3.8 Å from the van der Waals (vdW) surface of an atom accessible to the surface of the protein, and not assigned as a core cell. The ligand core is defined only to be any grid–cell closer than $\sqrt{1.5}$ of atoms not exposed to the protein surface.

I have adapted this function in two ways. First I have modified the core of the receptor to only include atoms that are not exposed to the surface. Doing this, preliminary results indicated much better ranking of the configurations, using only the shape complementarity measure.

Secondly, I have redefined the way to calculate surface accessibility. In this work, I have implemented a two-step algorithm that approximates solvent accessibility well. Given a protein and a probe radius of p;

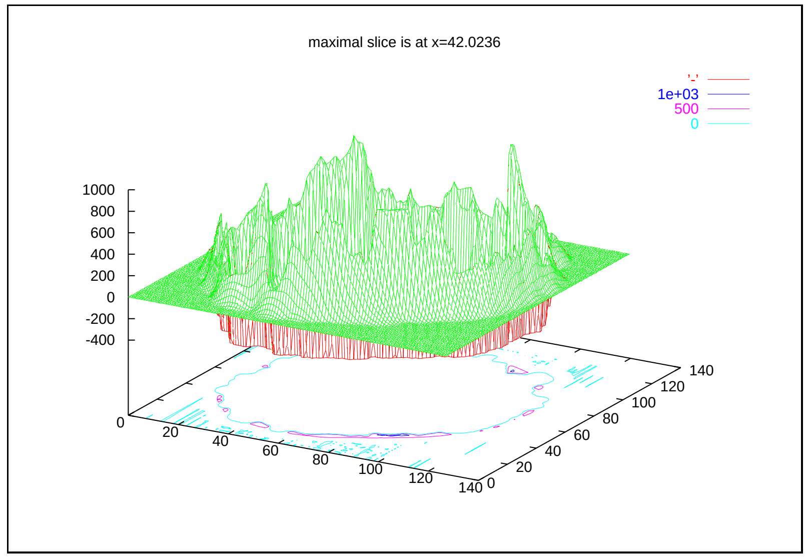


Figure 3.2: A slice of the shape complementarity correlation measure E_{SC}. Each point corresponds to a translation, and the value of that point is the shape complementarity measure for that translation.

Step 1 A grid with fixed resolution is placed to cover the protein in question. Every grid–cell that is within the probe radius p plus the vdW radius of any atom is assigned 1. Every other grid–cell is assigned 0.

Step 2 For every grid–cell assigned 1, but with neighbouring grid–cell assigned 0 (in all 26 directions of the grid), examine nearby atoms, and increase their surface accessibility, if they are within the probe length p plus the vdW radius (See Figure 3.3).

As can be seen in Figure 3.4 the surface assignment algorithm readily detects surface accessible atoms in a protein. It should also be noted that the algorithm is faster and more robust than that of Conolly et al [9], which failed to produce results on many of the test examples. My own method does however not produce results that can be used to directly measure exact surface accessibility, like that of Conolly, but it does a good job of assigning an approximate value to that effect.

To be able to implement the increment of the atoms surface accessibility in a fast way I use the space partition tree (See Section 3.1.2) to make queries as to which atoms are close to a given surface grid–cell. This is necessary, as the

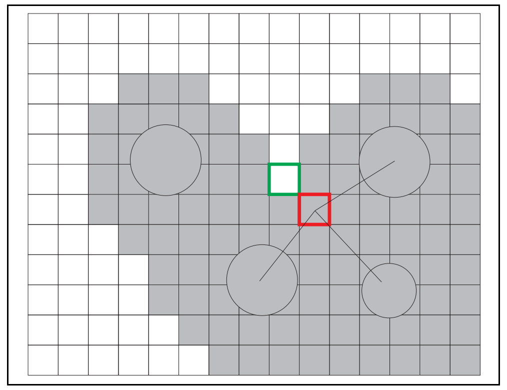


Figure 3.3: Example of grid–cell that has been assigned 1 (red) and have at least one neighbour that is assigned 0 (green), the nearby atoms surface accessibility is increased. (example is 2 dimensional for clarity, actual algorithm operates in 3 dimensions)

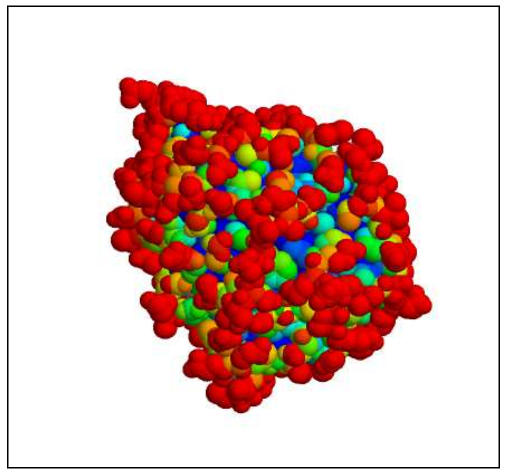


Figure 3.4: Example of surface assignment of a protein using the grid–based method. Red indicates exposed atoms, blue indicates buried atoms.

algorithm needs to do a query to nearby atoms on every grid–cell on the surface.

The interaction limit to the shape complementarity function used in the field function sampling is only 3.8 Å. In Figure 3.2 is an example of the shape complementarity correlation grid after application of the Fourier transforms. Note how some locations along the edge of the protein have better shape complementarity than others, shown as tops in the graph.

3.2.2 Desolvent effects

As it has also been shown that desolvent effects may be a driving force in the recognition of protein binding configurations, I have chosen to implement the ACE [1] energy measure of Zhang et al [34] which calculates atomic desolvation energies. The ACE measure establishes the desolvation free energy used to transfer an atom from water to a proteins interior. The method describes the energy as a difference between atoms in side-chain–side-chain contact and atoms in solution contact. The measure describes an energy matrix of 18 atom–atom contact energies that is to be summed up over all atoms within 6 Å of each other. Evaluating this measure for a given protein–protein interaction, and taking the sum of all atoms (a) in the receptor (R) that is within a distance ($r_a b$) of 6 Å of an atom (b) of the ligand (L), would presumably give the desolvation energy gained/lost from having that particular configuration rather than having the proteins separate (see Equation 3.8).

$$\Delta E_{DS} = \sum_{a \in R, b \in L} \mathrm{ACE}_{ab}, r_{ab} < 6. \tag{3.8}$$

However, as this function is not described as a correlation between two functions, it needs to be modified before it can be used in this method. The general problem is that it needs to be described as a correlation between two functions, similar to that of Equation 3.4. This is however not directly possible for functions of this type. Instead I have chosen to approximate it by finding values for the density and field functions that will minimized the error.

$$\Delta E'_{DS} = \sum_{a \in R, b \in L} \mathrm{ACER}_a * \mathrm{ACEL}_b, r_{ab} < 6. \tag{3.9}$$

Thus I need to minimize $E_{DS}(a,b) - E'_{DS}(a,b), a, b \in \text{atomtype}\{1..18\}$. I do this by simulated annealing, adjusting the two *complex* arrays of ACER and ACEL values by a series of small modifications. The end result has a correlation coefficient of 0.98 of the original ACE energy values, with an RMS deviation of 0.19 KJ/Mol.

[1] ACE: Atomic Contact Energy

This measure is then used to define the ACE density and field functions to be sampled in the grid. Using the approximated formula of Equation 3.9 I get;

$$
\begin{aligned}
R_{DS}(c) &= \sum_{a \in R} \mathrm{ACEL}_a, a \text{ in c} \\
L_{DS}(c) &= \sum_{b \in L} \mathrm{ACER}_b, r_{bc} < 6\text{Å}
\end{aligned}
\tag{3.10}
$$

Where a and b are atoms in the receptor (R) and ligand (L), and c is a given grid–cell coordinate (i, j, k) in the grid, and r_{bc} is the distance from the atom to the grid cell. And finally, using Equation 3.10 I define the derived grid–based desolvent energy measure as;

$$
E_{DS}(i,j,k) = \mathrm{Re}\Big[\sum_{l,m,n=1}^{N} R_{DS}(l,m,n) \times L_{DS}(l+i, m+j, n+k)\Big]
\tag{3.11}
$$

Using Equation 3.3 the desolvent correlation function E_{DS} can be calculated for all grid-cells using discreet Fourier transformations of the grids, after sampling the density and field functions R_{DS} and L_{DS} in two grids.

$$
E_{DS} = \frac{1}{N^3}\mathrm{IFT}(\mathrm{IFT}(R_{DS})\mathrm{DFT}(L_{DS})).
\tag{3.12}
$$

3.2.3 Electrostatic interactions

I have chosen to use a simple additive electrostatic interaction potential similar to that of Equation 2.1. However, some issues need to be resolved. First the distribution and value of point charges need to be addressed. I have chosen an atom–centered point–charge distribution, as this makes the transformation of the function into density and field function relatively simple. Secondly several atom–centered point–charge distributions exist, although I have only been able to publically get hold of the AMBER forcefield [11] point charges. These charges are supplied for individual residue types giving the possibility of distinguishing between charged and uncharged residue types, as well as distinguishing between the charges of the N-terminus, C-terminus and the in-chain type of residues. The second problem with the simple electrostatic field is that using the permittivity of vacuum is a bad choice since the environment can hardly be said to be vacuous. I here adopt the method of Gabb et al [13] and implements a permittivity that depends on the distance;

$$\epsilon(r_{ab}) = \begin{cases} 4 & r_{ab} \leq 6\text{Å} \\ 38r_{ab} - 224 & 6\text{Å} < r_{ab} < 8\text{Å} \\ 80 & r_{ab} \geq 8\text{Å} \end{cases} \tag{3.13}$$

Where a and b are two atoms, and r_{ab} is the distance between them. Using Equation 3.13 I can set up density and field functions, given I choose a distance cutoff for the electrostatics effects. I have chosen a cutoff of 12 Å, since at this point the interactions energies are insignificant. The R_{ES} function describes the electrical field of the charges around the grid-cell. The factor 1390.084 is a factor that converts the electrostatic energy into kilojoule per mole.The L_{ES} score is simply a delta-function, with the point charges of the atoms within it summed up.

Let q_a and q_b be point charges in atom a and b. Then;

$$\begin{aligned} R_{ES}(c) &= \sum_{a \in R} q_a, a \text{ in c} \\ L_{ES}(c) &= \sum_{b \in L} 1390.084 \frac{q_b}{\epsilon(r_{bc}) r_{bc}}, r_{bc} < 12\text{Å and } r_{bc} > vdW_k \end{aligned} \tag{3.14}$$

Where a and b are two atoms, c is a given grid–cell coordinate (i, j, k) in the grid, and r_{bc} is the distance between the atom b and the grid cell. In Equation 3.14 impose the limitation on the electrical fields, that they are 0 when within the vdW radius of any atom ($r_{bc} > vdW_k$). This is done to limit the matches that would call for atoms to overlap. In Figure 3.5 a slice of the sampled cells of the L_{ES} field function is shown, revealing distinct points of charge changes on the surface of the protein.

$$E_{ES}(i, j, k) = \text{Re}\Big[\sum_{l,m,n=1}^{N} R_{ES}(l, m, n) \times L_{ES}(l + i, m + j, n + k)\Big] \tag{3.15}$$

Using Equation 3.3 the electrostatic correlation function E_{ES} can be calculated for all grid-cells using discreet Fourier transformations of the grids, after sampling the field and density functions R_{ES} and L_{ES} in two grids.

$$E_{ES} = \frac{1}{N^3} \text{IFT}(\text{IFT}(R_{ES}) \text{DFT}(L_{ES})). \tag{3.16}$$

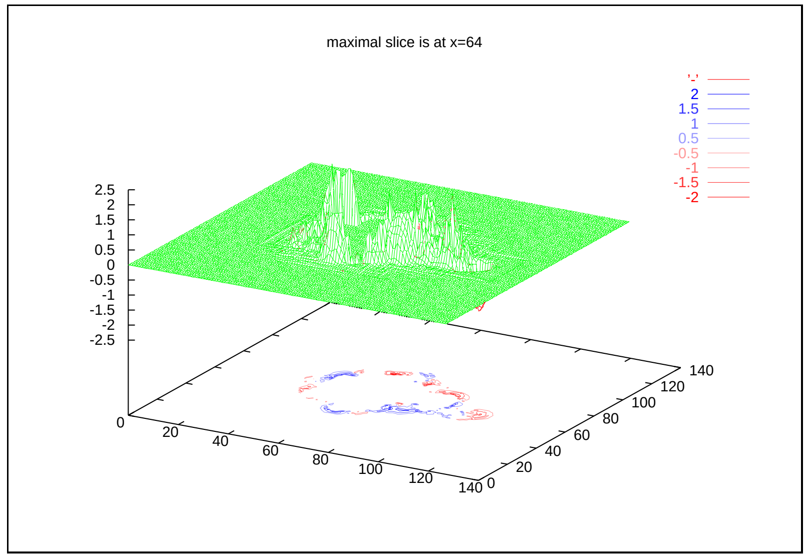


Figure 3.5: A slice of L_{ES} grid. Positive electrical fields are marked blue contours, while negative electrical field are marked red contours.

3.2.4 Energy function scaling

At this point, it becomes necessary to use some kind of scaling measure to arrive at a final energy correlation measure. Using Equations 3.16, 3.12 and 3.7 we arrive at a scaled energy correlation measure in Equation 3.17, that allows us to rank all translational conformations of the protein-protein interaction given the implemented energy considerations.

$$E_{total} = \alpha E_{SC} + E_{DS} + \beta E_{ES}. \tag{3.17}$$

Where α and β are scaling coefficients. These coefficients should be derived from comparative analysis of known, unbound protein–protein interfaces to choose the scaling factors that produce the best results. Preliminary tests using only a limited search have given these factors; $\alpha = 0.5$, $\beta = 0.0125$. These measures were found using only a single bound protein complex by minimizing the distance from one of the 16 best results to the correct translation.

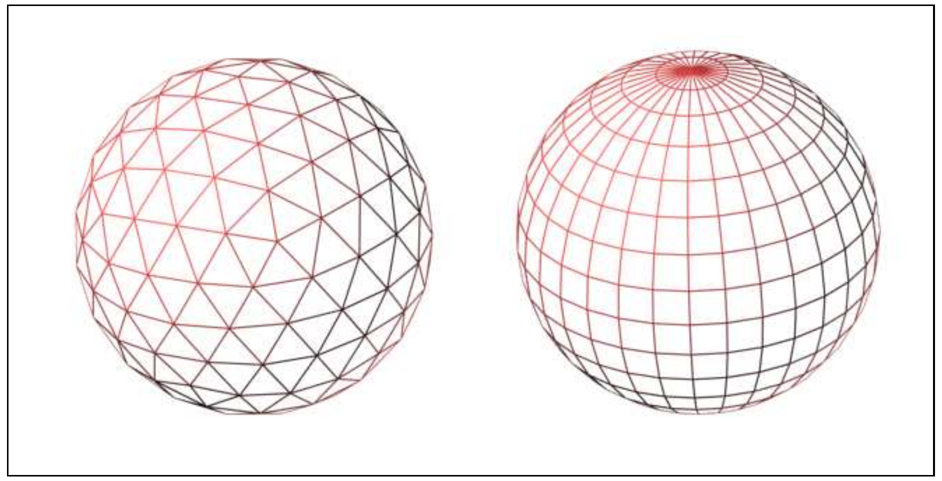


Figure 3.6: Different subdivision of rotational space. The right subdivision uses simple latitude/longitude subdivision, while the left uses a hierarchical subdivision of an icosahedron using triangulation of its triangular faces. Final rotational orientation is determined by the center of a face at the lowest subdivision.

3.3 Rotational space subdivision

As the algorithm need to search through all $O(n^6)$ [2] configurations of the rotational and translational space, it is important to make good choices in the subdivision of the space. If I had chosen a simple Eulerian subdivision of the rotational angles, the sampling of the rotational orientations would become uneven as angles close to the 'poles' of the sampling sphere would be samples much closer than angles at the 'equator' (see Figure 3.6 (right)). However if I create the sampling through the subdivision of an icosahedron (see Figure 3.6 (left)) the sampling becomes more even.

The figures only clearly show 2 of the 3 dimensions in the rotational space. To be able to subdivide the last rotational parameter on the same scale as the other two, one need to calculate a reasonable number of sections to subdivide the last dimension by.

The surface of the unit sphere circumscribing icosahedron constitutes 4π, subdividing two dimensions into 20 even segments. To find out how to initially subdivide the last angular dimension, you can use the average subdivision of the other two dimensions, which would then be; $2\pi/\sqrt{\frac{4\pi}{20}} = 7.92$ segments. Therefore an initial of 8 segments of the first subdivision of the last dimension of rotational space would match that of the 20 subdivisions of the sphere. Then, as the sphere segments are subdivided by triangulation into 4 subsegments, the last rotational

[2] assuming that rotational space is sample to the same degree as translational space

dimension is similarly subdivided into 2 subsegments. Thus the rotational subdivision starts at $20 \times 8 = 160$ segments, and subdivides into 1280 and then 10240 etc.

3.4 Simple protein–protein docking

At this point a simple protein–protein docking algorithm can be constructed, using the scaled correlation measures above.

```
Sample Receptor correlation grids R
Create inverse Fourier transformed Receptor correlation grids;
    R = IFT(R)
Sample Ligand correlation grids L
Loop though all angles.
    Create rotated Ligand correlation grids L_rot=rotate(L)
    Create Fourier transformed Ligand correlation grids;
        L_rot = IFT(L_rot)
    Calculate correlation grids
        E = 1/N^3 IFT(L_r * R)
    Save highest scoring translations in E
End Loop
```

This algorithm is very similar to that of Gabb et al [13], except it optimizes the process by avoiding the work of continuously sampling the ligand correlation functions for each rotation. When rotating the grids, I have to take special care when evaluating cells that have delta function values. As simple rotation of the grids may cause some cells not to be sampled in the rotated grid, I implement separate rotation function for delta function cells and for field function cells (e.g. R_{ES} is a delta correlation function, while L_{ES} is a field correlation function). Loosing a grid cell of a delta function when rotating would be equal to removing all atoms within that cell.

3.5 Result filtering

When locating the highest scoring translations within a correlation grid, you risk that several high–ranking candidates are ignored because of a single high peek, whose top contains all the top–scoring cells that you collect. This is however not very informative, and instead a set of clustered results is wanted, where such a peak would be shown as a single good solution. I have therefore devices a filtering method that cluster related high scoring translations within a grid.

The filtering method starts by sorting all the cells within the grid according to the final scaled correlation score. Then the highest score is selected and added to the pool of results. After this the location of the high scoring cell within the grid is located. A recursive walk is done marking the grid cells within 2.0 Å as being within an already reported result. Then all neighbouring cells (that has not already been marked) and which have a *descending* score gradient within the grid are subsequently marked as being within an already reported result (up until a distance of 8.0 Å). After the marking the next highest scoring cell in the sorted list, that has not been marked is selected, and so forth. This allows for a many very similar translational results to be reported as one. By following the descending score gradient I effectively marks all the grid cells in the *energy funnel* leading up to the high scoring grid cell. However lower scoring local minima are allowed to be reported as well (if more than 2.0 Å away) as they would not be covered by a path along a descending energy gradient.

3.6 Bounds of rotational space

As the algorithm searches through the rotational space, it requires that the energy correlation function of $O(n^3)$ rotational configurations to be optimized. To speed this up, I have utilized the hierarchical subdivision of the rotational space to produce a branch–and–bound algorithm, potentially reducing the needed search space.

However to implement this, all correlation functions need to be modified to be able to calculate correlation bounds for several sampled density and field functions.

Given a set of density and field functions we need to define ways to sample these functions into grids, and then be able to produce a correlation function that gives us a bound on their best possible configuration. To implement bounds of rotational space we only need to consider sampling multiple grids of the Ligand density and field functions, as it is the only element being altered when searching through the rotational space.

When the bounds are in place, a branch–and–bound algorithm may be implemented that recursively searches through the hierarchical subdivision of rotational space, and at each branch evaluates a bound for the best scores possible below that point. If the best bound in the resulting correlation grid is worse than the worst result we have yet saved as interesting, the entire subsection of the rotational space may be skipped.

The method is however sensitive to the data. If reasonably high-scoring conformation are found for almost all rotational configurations, the method does not limit the amount of work needed.

Presently the implementation of the branch–and–bound search is not yet finished.

3.6.1 Bound on shape complementarity

Here i will define the bound correlation functions for the shape complementarity measure.

Let $L_{SC,i}$ be be a set of k density grids, $i \in \{1..k\}$, whose correlation bound against R_{SC} is sought.
Let the grid $L_{SC,max}$ be defined as;

$$\begin{aligned} Re[L_{SC,max}] &= max_{i\in\{1..k\}} Re[L_{SC,i}] \\ Im[L_{SC,max}] &= max_{i\in\{1..k\}} Im[L_{SC,i}] \end{aligned}$$

Equivalently $L_{SC,min}$ is defined. Since both real and imaginary part of $L_{SC,i}$ are positive, I can now define the bound correlation function for shape complementarity as;

$$\begin{aligned} E_{SC,bound}(i,j,k) &= \mathrm{Re}\Big[\sum_{l,m,n=1}^{N} R_{SC}(l,m,n) \times L_{SC,max}(l+i,m+j,n+k)\Big] - \\ &\quad \mathrm{Im}\Big[\sum_{l,m,n=1}^{N} R_{SC}(l,m,n) \times L_{SC,min}(l+i,m+j,n+k)\Big] \end{aligned} \tag{3.18}$$

Similarly to Equation 3.7, using Equation 3.18 and Equation 3.3 we can now define the bound correlation function as;

$$\begin{aligned} E_{posSC} &= \frac{1}{N^3}\mathrm{IFT}(\mathrm{IFT}(R_{SC})\mathrm{DFT}(L_{SC,max})). \\ E_{negSC} &= \frac{1}{N^3}\mathrm{IFT}(\mathrm{IFT}(R_{SC})\mathrm{DFT}(L_{SC,min})). \\ E_{SC,bound} &= Re[E_{posSC}] - Im[E_{negSC}] \end{aligned} \tag{3.19}$$

Note that the bound requires an extra grid to be defined. Trivially for $k = 1$ $L_{SC,max}$ will be equal to $L_{SC,min}$, being the result of the unbounded case.
The grid $E_{SC,bound}$ will now hold the bounds for the correlation function E_{SC} in Equation 3.7 given a sampling of multiple ligand density functions.

3.6.2 Bound for desolvent effects

As the desolvation effect field and density grids can have both positive and negative values in both the real and imaginary parts of their complex value, creating a bound for this measure proved very hard. I opted instead to redefine the desolvation energies by adapting the method of Chen et al [7] that defines the field as an average desolvation effect $\mathrm{ACE}_{a,avg}$ for a given atom type a, and the density function as 1.

$$
\begin{aligned}
Im[R_{DS}(c)] &= \sum_{a \in R}^{a \text{ in c}} 1 \\
Re[L_{DS}(c)] &= \sum_{a \in R}^{r_{ac} < 6\mathring{A}} \mathrm{ACE}_{a,avg} \\
Im[L_{DS}(c)] &= \sum_{b \in L}^{b \text{ in c}} 1 \\
Re[L_{DS}(c)] &= \sum_{b \in L}^{r_{bc} < 6\mathring{A}} \mathrm{ACE}_{b,avg}
\end{aligned}
\tag{3.20}
$$

Where a and b are two atoms in the receptor R and ligand L, c is a given grid–cell coordinate (i, j, k) in the grid, and r_{bc} is the distance between the atom b and the grid cell. Then defining the correlation function to be;

$$
E_{DS}(i,j,k) = \mathrm{Im}\Big[\sum_{l,m,n=1}^{N} R_{DS}(l,m,n) \times L_{DS}(l+i, m+j, n+k)\Big] \tag{3.21}
$$

This allows us to create bounds in a similar way to the shape complementarity correlation bounds;

Let $L_{DS,i}$ be a set of k density grids, $i \in \{1..k\}$, whose correlation bound against R_{DS} is sought. Let the grid $L_{DS,max}$ be defined as;

$$
\begin{aligned}
Re[L_{DS,max}] &= max_{i \in \{1..k\}} Re[L_{DS,i}] \\
Im[L_{DS,max}] &= max_{i \in \{1..k\}} Im[L_{DS,i}]
\end{aligned}
$$

Since both imaginary part of $L_{DS,i}$, as well as R_{DS} is positive, I can now define the bound correlation function for desolvent complementarity as;

$$
E_{DS,bound}(i,j,k) = \mathrm{Im}\Big[\sum_{l,m,n=1}^{N} R_{DS}(l,m,n) \times L_{DS,max}(l+i, m+j, n+k)\Big] \tag{3.22}
$$

Similarly to previous transformations in Equation 3.12 this can be described using Fourier correlation theorem;

$$E_{DS,bound} = Im[\frac{1}{N^3}\mathrm{IFT}(\mathrm{IFT}(R_{DS})\mathrm{DFT}(L_{DS,max})] \tag{3.23}$$

The grid $E_{DS,bound}$ will now hold the bounds for the correlation function E_{DS} in Equation 3.12 given a sampling of multiple ligand density functions.

3.6.3 Bound for electrostatic interactions

As the electrostatic interaction deals with both positive and negative numbers, creating a bound requires a somewhat different approach than in the other cases. Here we sample the receptor field differently;
Let q_a and q_b be the point charges of atoms a and b.

$$\begin{aligned}
R_{tmp}(c) &= \sum_{a \in R} q_a, a \text{ in c} \\
Re[R_{ES}(c)] &= R_{tmp}(c), \text{iff} R_{tmp}(c) \text{is positive} \\
Im[R_{ES}(c)] &= R_{tmp}(c), \text{iff} R_{tmp}(c) \text{is negative} \\
L_{ES}(c) &= \sum_{b \in L} 1390.084 \frac{q_b}{\epsilon(r_{bc})r_{bc}}, r_{bc} < 12\text{Å and } vdW_k < r_{bc}
\end{aligned} \tag{3.24}$$

Where a and b are two atoms in the receptor R and ligand L, and c is a given grid–cell coordinate (i, j, k) in the grid, and r_{bc} is the distance between the atom b and the grid cell c.

Let $L_{ES,i}$ be be a set of k field grids, $i \in \{1..k\}$, whose correlation bound against R_{ES} is sought.
Let the grid $L_{ES,high}$ be defined as;

$$\begin{aligned}
Re[L_{ES,high}] &= min_{i \in \{1..k\}} Re[L_{ES,i}], Re[L_{ES,i}] \leq 0 \\
Im[L_{ES,high}] &= max_{i \in \{1..k\}} Re[L_{ES,i}], Re[L_{ES,i}] \geq 0
\end{aligned}$$

Similarly let the grid $L_{ES,min}$ be defined as;

$$\begin{aligned}
Re[L_{ES,low}] &= min_{i \in \{1..k\}} Re[L_{ES,i}], Re[L_{ES,i}] \geq 0 \\
Im[L_{ES,low}] &= max_{i \in \{1..k\}} Re[L_{ES,i}], Re[L_{ES,i}] \leq 0
\end{aligned}$$

The grid $L_{ES,high}$ will now hold the highest sampled negative charge of all the sampled $L_{ES,i}$ in the real part, while holding the highest sampled positively charge in the imaginary part. Similarly $L_{ES,low}$ now holds the weakest sampled positive charge in its real part and the weakest sampled negative charge in its imaginary part.

To create a bound for a given electrostatic grid interaction, you need to calculate the highest attractive force minus the lowest repulsive force.

$$\begin{aligned} E_{bound} &= Im[L_{ES,high}]Re[R_{ES}] + Re[L_{ES,high}]Im[R_{ES}] \\ &\quad -(Re[L_{ES,low}]Im[R_{ES}] + Im[L_{ES,low}]Re[R_{ES}]) \end{aligned}$$

This would get the strongest sampled attraction minus the weakest sampled repulsion. We can now define the correlation function as;

$$\begin{aligned} E_{ES,bound}(i,j,k) &= \mathrm{Im}\Big[\sum_{l,m,n=1}^{N} R_{ES}(l,m,n) \times L_{DS,high}(l+i, m+j, n+k)\Big] - \\ &\quad \mathrm{Im}\Big[\sum_{l,m,n=1}^{N} R_{ES}(l,m,n) \times L_{DS,low}(l+i, m+j, n+k)\Big] \end{aligned} \tag{3.25}$$

This would return the most attractive force minus the least repulsive force of all the interactions, which is a bound for the most attractive force. Using Equation 3.25 with Equation 3.3 this correlation can be described by discreet Fourier transformations of the grids;

$$\begin{aligned} E_{att} &= \frac{1}{N^3}\mathrm{IFT}(\mathrm{IFT}(R_{ES})\mathrm{DFT}(L_{ES,high})) \\ E_{rep} &= \frac{1}{N^3}\mathrm{IFT}(\mathrm{IFT}(R_{ES})\mathrm{DFT}(L_{ES,low})) \\ E_{ES,bound} &= Im[E_{att}] - Im[E_{rep}] \end{aligned} \tag{3.26}$$

The grid $E_{ES,bound}$ will now hold the bounds for the correlation function E_{ES} in Equation 3.16 given a sampling of multiple ligand density functions.

3.6.4 Sampling of rotational density and field bounds

To be able to use the rotational subdivision the rotational search needs to calculate all rotational grids below a certain level, and then merge them using the bounded field functions above. If the bound does not limit the search, the algorithm needs to evaluate the bounded field functions for the rotational space below, etc. To limit the amount of work done, the rotation and merging of the ligand field grids are done recursively. The intermediate results are stored down to the grids at the bottom of the recursion. This does require a lot of memory, but an effective clean-up strategy makes sure it is released when unneeded. At the chosen depth of recursion at most 72 grids need to be stored at a time.

Given a tree $\mathcal{R}$ of rotation subdivision as defined in Section 3.3, Let $\mathcal{R}$.rot be the bound field and density functions for all rotational subdivision of $\mathcal{R}$ given the Ligand L, then;

```
Sample(RotationTree R, Ligand L)
    If ( R.rot exist )(
        Return R.rot
    Else
        Create R.rot
        If ( isleaf(R) )
            R.rot = rotate(L.grid,R.angle)
        Else
            Sample(R.child[i],L), i=1..8
            Merge(R.rot,R.child[i],rot)
        End
    End
```

Where the function *Merge()* will merge the field and density bound grids as described in this section.

3.6.5 Branch–and–bound algorithm on rotational space

The docking algorithm can now uses the same branch–and–bound algorithm. Given a tree $\mathcal{R}$ of rotation subdivision, and a receptor R, with density and field grids R.grid, we can describe this algorithm;

```
Docking(RotationTree R, Receptor R, Ligand L, Results Res)
    Sample(R, L)
    If (isleaf(R))
        Return Evaluate(R.grid,R.rot);
    Else
        Tmp = Evaluate(R.grid,R.rot);
        If ( bestresult(Tmp) better than worstresult(Res) )
            Loop over all children i:
            Res = Res ∪ Docking(R.child[i],R,L,Res)
        End
    End
```

Here the function *Evaluate()* will evaluate the grids by discreet Fourier analysis, and return the top filtered results from the ranking done by the bounded energy functions. The union of results will only keep a limited number of the best ranked results.

3.7 Semi–flexible docking

As the algorithm also need to explicitly take into account the flexibility of the residue side-chains, I propose to let the ranking of the possible configurations be determined by a bound for the best possible energy that a side-chain configuration may give. This requires however that yet another set of correlation bounds be set up, that also makes it possible to sample alternative field grids for the receptor.

The energy bounds were found in a similar way to that of Section 3.6, except the bounding of the receptor density functions were also needed.

3.7.1 Sampling of side-chain energy bounds

The possible side-chain conformation are sampled from Bower et al's rotamer library [5], which is a collection of side-chain configurations dependent on the backbone configuration. The rotamer library gives the side-chain configuration as a set of chi-angles (see Section 2.2.2). Therefore I have implemented a program that builds up every atom of a side-chain (and its part of the backbone) given the phi/psi and chi-angles. This program was used in conjunction with the energy bounds of Section 3.7 to produce small field and density bound grids for every single side-chain type, where the Cα,C and N atoms for each instance was kept in the same positions in the grid. The grids could then be saved for fast access.

To build the density and field function of the receptor or ligand, the grids for each residue in the protein was in turn merged with the density and field grid for the protein. The grid was rotated and translated to fit the current residue being sampled.

3.8 Database search algorithm

To extend the method into searching against a large database, I propose to cluster the proteins in the database according to structure, such that similar proteins are grouped together.

After finding an optimal configuration where the fields are the most alike, the energy fields can then be merges similar to the merging used in Section 3.7. The clusters are then grouped together to form still larger clusters, where there exist reasonably high similarity between the fields.

To search a given cluster of proteins, the search algorithm then evaluate the representative field and density of a cluster relative to the query structure to get a bound on how well the individual proteins in the cluster can score. If the best result is worse than some user defined limit for good interaction energy, none of the proteins in the cluster needs to be examined separately. This technique has particular interest in the case where the structures have been resolved with NMR. In this case several samples of the proteins configurations are usually made, which gives an idea of which parts of the protein are flexible, as these parts will have different configurations in the samples. With this technique, the configurations can be clustered together. The method can then rank the binding configurations taking into account the various *backbone* movement present in the NMR data.

Chapter 4

Evaluation

As the method has not yet been finished, it has not been possible to evaluate it. I do however have one example of the power of the method. As a part of testing the Fourier correlation methods, the shape complementarity measure, and the result filtering method, i determined the optimal configuration for a bound protein complex. The result was very promising, as the method turned out a very near native configuration as it number 1 candidate (see Figure 4.1).

Other key points of the algorithm need to be addressed in any evaluation of the method.

First, the methods various algorithms need to be evaluated for effectiveness. Especially the branch–and–bound algorithm applied to reducing the rotational space needs to be evaluated to see if the bounds are effective at reducing the search space.

The methods sensitivity also needs to be evaluated. If the method is to be used to scan a database for possible interactions, it is important to know how many false positive solutions it comes up with. Especially with the introduction of bounds in the method, one would assume that using the bound as a ranking factor would cause more false positive bindings. This is a point that is often overlooked in evaluation of the protein–protein docking algorithms currently developed. Here the ability to let the algorithm state that two proteins does not interact is not even considered.

When reporting protein–protein docking algorithm performance often only a list of candidates that is close to the correct binding configuration is reported along with at what rank they figure in the list of possible results.

This leads to the problem of developing a way to evaluate the results. In the case of discreet Fourier docking programs, the docking configurations are often simply ranked, meaning that the success of the method is to be able to rank results close to the correct configuration among the highest ranged candidates. However, you need to determine if a configuration that is close the the correct one but ranked very low is better than a configuration a bit removed from the correct one, but highly ranked.

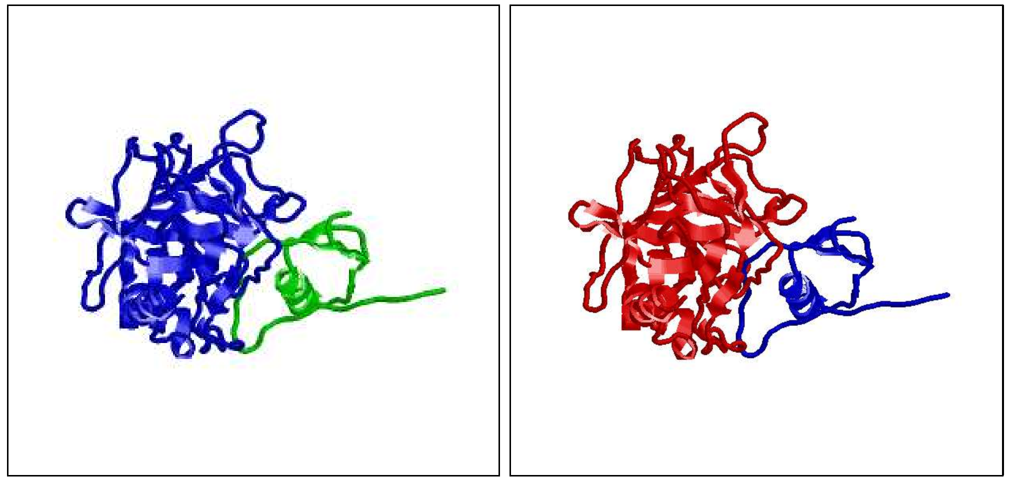


Figure 4.1: Evaluation of docking with shape complementarity only. Left; the correct bound configuration of the receptor and ligand. Right the highest ranking configuration.

What i have proposed to do, and implemented in the evaluation of the scaling factors, is to treat all the 16 top ranked configurations as equally important. My assumption is then, that any configuration not among the to 16, has failed to be predicted as a probable binding configuration. The measure of the methods ability to produce correct results is then the minimal distance between the the any of the top 16 results and the correct solution. The number 16 is arbitrarily chosen in this context, but not more arbitrary than any other cutoff in literature that i have found.

Another method would be to report the rank of a configuration sufficiently close to be accepted as a good result. Here a similar approach to above is made, except the validation measure is defined as the highest rank among the solutions that are acceptable as 'near-native'. This method is similar to that of Chen it al [7], except that the rank is here to be used as the absolute measure of quality.

4.1 Data collection

To be able to do evaluation on the results of any algorithm, I have to collect a good evaluation dataset. Since my algorithm will only work on proteins of known tertiary structure, I need a set of proteins where I know both the tertiary structure of the individual proteins, as well as the tertiary/quaternary structure of the complexes. An obvious choice is to look in the Brookhaven Protein Databank (PDB) [4] as it is a continually updated database that is publically available. However, not all entries of the database are suited to use in a test set, and there is also the problem of identifying the complexes as well as their separate unbound components (if represented) in the database. As a result of the Conference "Modeling of Protein Interactions in Genomes", Charleston, June 16–19, 2001 an initiative was made by the participants to construct a dataset for the very purpose of evaluating protein docking algorithms. The dataset was developed by Dr. Zhiping Weng's group.

4.2 Dataset validation

Before any datasets can be used by the program, it is important to perform a number of checks and automated alterations to the data.

4.2.1 Hydrogen atom positioning

One common problem with PDB entries derived from X-ray crystallography and NMR is that these techniques can only distinguish heavy atoms. Therefore the assignments of hydrogen atoms are often missing from the database files. As some of the energy measures in the later stages of the method relies on hydrogen atoms, I created a tool to assign hydrogen atom positions from the relative conformation of the residue atoms. The system is able to assign hydrogen atoms to a variety of different form of residues, including special cases of histidine as well as handle the specifics of the C–terminus and N–terminus for all residues. In all 87 different states of residues are covered and definitions to add over 200 hydrogen atom configurations are done. In the cases where hydrogen atoms have a degree of freedom in movement an arbitrary position has been chosen. The protein in Figure 2.5 is an example of a protein that has added hydrogen atoms.

4.2.2 Ambiguous assignment

As mentioned above X-ray crystallography and NMR is only able to distinguish heavy atoms, but furthermore the methods are unable to distinguish between some atoms, such as nitrogen and carbon. In the case of side-chains such as Histidine, this causes the assignment of orientation to be ambiguous (see Figure 4.2) and can potentially be rotated 180°. While some structures have made correct assignments, i should not rely on all having the correct assignment.

At the current stage of development I do not optimize the hydrogen rotational placement, nor do I check the histidine ambiguity that is possibly in error in the supplied entries. However several of the tools included in GROMACS [3, 17] molecular dynamics package could be put to use to do this prior to calculation.

In the cases where the supplied structure have more than one alternative for residue assignment, or atomic positioning, the first assignment is chosen.

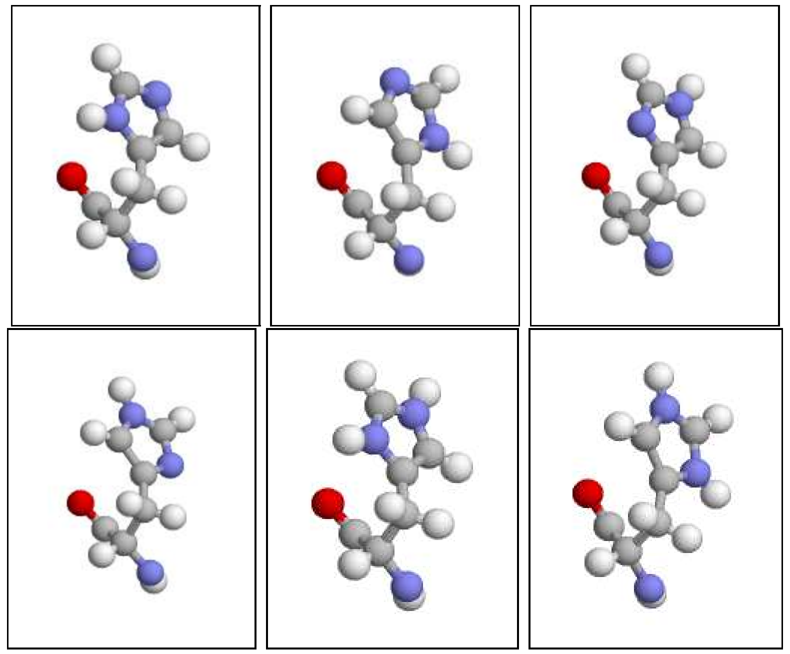


Figure 4.2: The ambiguous assignment of the atoms of histidine. As the structural determination is unable to distinguish between carbon (grey) and nitrogen (blue) and hydrogens, determining the orientation of the histidine may be erroneous. Likewise several variants of hydrogen assignments (and charge) also exist for histidine.

4.2.3 Energy minimization

In the cases where the test set only contains 3d structure of the bound complex but not of the unbound configuration as one would expect them to be with the intended use of the programme, i need a procedure that would produce a result that would be 'honest' in respect to not knowing the exact configuration of the protein in the final complex. In this case i reassigned the side-chain conformations using the SCRWL package of Bower et al. [5]. This was the case with 23 of the 102 examples from Chen et al. [7]. This step ensures that the results of that of the algorithm on bound proteins are comparable to that of unbound proteins.

4.2.4 Unbound protein alignment

As I need to be able to tell if the algorithm correctly identifies the binding configuration of the bound complexes given in the dataset, i need to determine what that configuration is. As the bound proteins and the unbound proteins has different configurations both internally and with respect to the rotational and translational configuration, I need to find a good fit that describes the unbound proteins relative position to the bound proteins. To do this I employ the CE [1] [29] tool. The program can structurally align two protein structures against each other, returning the translation and rotation that optimally translates one protein onto another. Using this tool, i find translational and rotational configurations needed to transform the unbound receptor into the bound receptor, as well as transform the unbound ligand into the bound ligand, and then combine the two to form a complex where the receptor and ligand are of the unbound state. These complexes are then read into my docking program where their relative configuration can be determined as the 'correct' mode of association, and thereby used to evaluate the results from the docking algorithm.

4.3 Further development

Even though the project is not in a finished form, I still have thought about some improvements to the algorithm. One of the problems is the frequent use of the discreet Fourier transform. I am convinced that all except one Fourier transform could be moved outside the loop that traverses the rotational space. The reason for this belief is that I can see that the rotation I do of the ligand field and density grids could be done in Fourier space. In fact according to Owen [23] rotation in Fourier space is a way to ensure superior quality. Thus, a cartesian rotation and a discreet Fourier transform would be replaced by a single Fourier space rotation for each ligand field density grid. Also the scaling and combination of the resulting energy functions could also be done in Fourier space, leaving only a single discreet

[1] Protein structure alignment by incremental combinatorial extension

inverse Fourier transform in each instance of the loop that traverses the rotational space.

Also, the alternative structures in the PDB files that are currently ignored (see Section 4.2.2) could with the flexible side-chain bounds in place become part of the search, by simply sampling all the alternatives of the protein structure, arriving at an energy bound for all possible alternatives.

Chapter 5

Conclusions

The project has not been finished fully, and I must admit that the size of the project was bigger than I imagined. Not all energy bounds has been implemented, nor has the scaling parameters of Section 3.2.4 been evaluated fully using the dataset of Chen et al. Thus neither the branch–and–bound, the flexible side-chain bound, nor the database search algorithms are in place at this time. Currently the source code developed for the project take up more than 15.000 lines of C++.

On the positive side, the preliminary docking attempts using only the modified, soft, shape complementarity measure was able to rank a near-native correct bound configuration at the very top (out of 2097152 possible), considering only the translational degrees of freedom.

All modules pertaining to the manipulation of atoms and proteins, as well as simple grid manipulation such as rotation, multiplications discreet Fourier transformation etc, are done, so is a number of support libraries handling complex math and 3 and 4 dimensional vector and matrix manipulation such as rotation, quaternions and coordinate system changes. The space partitioning tree related to the grid sampling, as well as the several alternative grid sampling algorithms have also been finished. The surface accessibility function has also been finished. Also the static docking field grids are in place, and could potentially be evaluated at this point. The test set data collection and its preparations are completed for evaluation.

I also believe that the general libraries of the whole system that has been developed are highly modular, and will serve as a good platform for further research in the area. Also i have been exchanging ideas with Dr. Rong Chen of Boston University.

List of Figures

Bibliography

[1] C. Andersen, *Neural network assignment of protein secondary structure with increased predictability*, Master thesis, The technical University Of Denmark, Lyngby, Denmark (1998).

[2] D. Bacon and J. Moult, Docking by least-squares fitting of molecular surface patterns, *J Mol Biol* **225(3)** (1992), 849–858.

[3] H. Berendsen, D.van der Spoel, and R.van Drunen, Gromacs: A message-passing parallel molecular dynamics implementation, *Comp Phys Comm* **91** (1995), 43–56.

[4] F. Bernstein, T. Koetzle, G. Williams, E. Meyer, M. Brice Jr., J. Rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi, The protein data bank: A computer-based archival file for macromolecular structures, *J. Mol. Biol.* **112** (1977), 535–542.

[5] M. Bower, F. Cohen, and R. Dunbrack Jr, Prediction of protein side-chain rotamers from a backbone-dependent rotamer library: A new homology modeling tool, *J Mol Biol* **267** (1997), 1268–1282.

[6] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, and M. Karplus, Charmm: a program for macromolecular energy, minimization and dynamics calculations, *J Comp Chem* **4** (1983), 187–217.

[7] R. Chen and Z. Weng, Docking unbound proteins using shape complementarity, desolvation, and electrostatics, *PROTEINS: Structure, Function, and Genetics* **47** (2002), in press.

[8] H. Claussen, C. Buning, M. Rarey, and T. Lengauer, Flexe: efficient molecular docking considering protein structure variations, *J Mol Biol* **308(2)** (2001), 377–95.

[9] M. Connolly, The molecular surface package, *J Mol Graphics* **11** (1993), 139–141.

[10] L. Conte, C. Chothia, and J. Janin, The atomic structure of protein-protein recognition sites, *J Mol Biol* **285(5)** (1999), 2177–98.

[11] W. Cornell, P. Cieplak, C. Bayly, I. Gould, K. J. Merz, D. Ferguson, D. Spellmeyer, T. Fox, J. Caldwell, and P. Kollman, A second generation force field for the simulation of proteins and nucleic acids, *Journal of the American Chemical Society* **117** (1995), 5179–5197.

[12] R. Dunbrack Jr and M. Karplus, Backbone-dependent rotamer library for proteins: Application to side-chain prediction, *J Mol Biol* **230** (1993), 543–574.

[13] H. Gabb, R. Jackson, and M. Sternberg, Modelling protein docking using shape complementarity, electrostatics and biochemical information, *J Mol Biol* **272** (1997), 106–120.

[14] J. Garavelli, The resid database of protein structure modifications: 2000 update, *Nucleic Acids Research* **28** (2000), 209–211.

[15] M. Helmer-Citterich and A. Tramontano, Puzzle: a new method for automated protein docking based on surface shape complementarity, *J Mol Biol* **235(3)** (1994), 1021–31.

[16] F. Jiang and S. Kim, "soft docking": matching of molecular surface cubes, *J Mol Biol* **219(1)** (1991), 79–102.

[17] E. Lindahl, B. Hess, and D.van der Spoel, Gromacs 3.0: A package for molecular simulation and trajectory analysis, *J Mol Mod* **7** (2001), 306–317.

[18] M. Meyer, P. Wilson, and D. Schomburg, Hydrogen bonding and molecular surface shape complementarity as a basis for protein docking, *J Mol Biol* **264** (1996), 199.

[19] J. Mills, T. Perkins, and P. Dean, An automated method for predicting the positions of hydrogen-bonding atoms in binding sites, *J Comput Aided Mol Des* **11(3)** (1997), 229–42.

[20] M. Mizutani, N. Tomioka, and A. Itai, Rational automatic search method for stable docking models of protein and ligand, *J Mol Biol* **243(2)** (1994), 310–26.

[21] R. Norel, S. Lin, H. Wolfson, and R. Nussinov, Molecular surface complementarity at protein-protein interfaces: the critical role played by surface normals at well placed, sparse, points in docking, *J Mol Biol* **252(2)** (1995), 263–73.

[22] R. Norel, D. Petrey, H. Wolfson, and R. Nussinov, Examination of shape complementarity in docking of unbound proteins, *Proteins* **36(3)** (1999), 307–317.

[23] C. B. Owen, High quality alias free rotation, *Proceedings of 30th Asilomar Conference on Signals, Systems and Computers* (1996).

[24] P. Palma, L. Krippahl, J. Wampler, and J. Moura, Bigger: a new (soft) docking algorithm for predicting protein interactions, *Proteins* **39(4)** (2000), 372–84.

[25] E. Pletneva, A. Laederach, D. Fulton, and N. Kostic, The role of cation-pi interactions in biomolecular association. design of peptides favoring interactions between cationic and aromatic amino acid side chains, *J Am Chem Soc* **123(26)** (2001), 6232–45.

[26] M. Rarey, B. Kramer, and T. Lengauer, Docking of hydrophobic ligands with interaction-based matching algorithms, *Bioinformatics* **15(3)** (1999), 243–250.

[27] M. Rarey, B. Kramer, T. Lengauer, and G. Klebe, A fast flexible docking method using an incremental construction algorithm, *J Mol Biol* **261(3)** (1996), 470–89.

[28] D. Ritchie and G. Kemp, Protein docking using spherical polar fourier correlations, *Proteins* **39(2)** (2000), 178–94.

[29] I. Shindyalov and P. Bourne, Protein structure alignment by incremental combinatorial extension (ce) of the optimal path., *Protein Engeneering* **9** (1998), 739–747.

[30] J. Taylor and R. Burnett, Darwin: a program for docking flexible molecules, *Proteins* **41(2)** (2000), 173–91.

[31] N. Tomioka and A. Itai, Green: a program package for docking studies in rational drug design, *J Comput Aided Mol Des* **8(4)** (1994), 347–66.

[32] I. Vakser and C. Aflalo, Hydrophobic docking: a proposed enhancement to molecular recognition techniques, *Proteins* **20(4)** (1994), 320–9.

[33] J. Yang and C. Kao, A family competition evolutionary algorithm for automated docking of flexible ligands to proteins, *IEEE Trans Inf Technol Biomed* **4(3)** (2000), 225–37.

[34] C. Zhang, G. Vasmatzis, J. Cornette, and C. DeLisi, Determination of atomic desolvation energies from the structures of crystallized proteins, *J Mol Biol* **47** (2002), 707–726.

Chapter 6

Appendices

6.1 A: Software packages and libraries used

RasMol 2.7.0	Molecular Graphics Visualization Tool
FFTW-2.1.3	A collection of fast C routines to compute the discreet Fourier Transform in one or more dimensions.
libpdb++	Library for parsing and writing files with 3d data of molecules in the PDB format.
GROMACS	Molecular Dynamics simulator energy minimization tools package http://www.gromacs.org/.
Rotamer Library	A backbone dependent rotamer library ([12]).
SCWRL v2.9	A side-chain configuration prediction tool ([5]).
CE	Protein Structure Alignment tool. ([29]).

6.2 B: Structure of validation dataset

* unbound/bound protein complexes. All others are unbound/unbound protein complexes

Complex	Receptor	Ligand	Description
1CGI	1CHG	1HPT	Alpha–chymotrypsinogen / Pancreatic secretory trypsin inhibitor
1CHO	5CHA	2OVO	Alpha–chymotrypsin / Ovomucoid 3rd Domain
2PTC	2PTN	6PTI	Beta–trypsin / Pancreatic trypsin inhibitor
1TGS	2PTN	1HPT	Trypsinogen / Pancreatic secretory trypsin inhibitor
2SNI	1SUP	2CI2	Subtilisin Novo / Chymotrypsin inhibitor 2
2SIC	1SUP	3SSI	Subtilisin BPN / Subtilisin inhibitor
1CSE	1SCD	1ACB(I)	Subtilisin Carlsberg / Eglin C
2KAI	2PKA	6PTI	Kallikrein A / Trypsin inhibitor
1BRC	1BRA	1AAP	Trypsin / APPI
1PPE*	2PTN	1PPE	Trypsin / CMT–1
1TAB*	2PTN	1TAB	Trypsin / BBI
1ACB	5CHA	1CSE(I)	Alpha–chymotrypsin / Eglin C
1BRS	1A2P	1A19	Barnase / Barstar
1MAH	1MAA	1FSC	Mouse Acetylcholinesterase / inhibitor
1UGH	1AKZ	1UGI	Human Uracil–DNA glycosylase / inhibitor
1UDI*	1UDH	1UDI	Virus Uracil–DNA glycosylase / inhibitor
1STF*	1PPN	1STF	Papain / Stefin B
2TEC*	1THM	2TEC	Thermitase / Eglin C
4HTC*	2HNT	4HTC	Alpha–Thrombin / Hirudin
1DFJ	2BNH	7RSA	Ribonuclease inhibitor / Ribonuclease A
1FSS	2ACE	1FSC	Acetylcholinesterase / Fasciculin II
1AVW	2PTN	1BA7	Trypsin / Sotbean Trypsin inhibitor

Table 6.1: Enzyme/inhibitor Complexes

Complex	Receptor	Ligand	Description
1MLC	1MLB	1LZA	IgG1 D44.1 Fab fragment / Lysozyme
1FBI*	1FBI	1HHL	IgG1 Fab fragment / Lysozyme
1WEJ	1QBL	1HRC	IgG1 E8 Fab fragment / Cytochrome C
1AHW	1FGN	1BOY	Antibody Fab 5G9 / Tissue factor
2JEL*	2JEL	1POH	Jel42 Fab Fragment / A06 Phosphotransferase
1BQL*	1BQL	1DKJ	Hyhel – 5 Fab / Lysozyme
1DQJ	1DQQ	3LZT	Hyhel – 63 Fab / Lysozyme
1JHL*	1JHL	1GHL	IgG1 Fv Fragment / Lysozyme
1BVK	1BVL	3LZT	Antibody Hulys11 Fv / Lysozyme
1NCA*	1NCA	7NN9	Fab NC41 / Neuraminidase
1NMB*	1NMB	7NN9	Fab NC10 / Neuraminidase
1MEL*	1MEL	1LZA	Vh Single–Domain Antibody / Lysozyme
2VIR*	2VIR	2VIU	Igg1–lamda Fab / Hemagglutinin
1EO8*	1EO8	2VIU	Bh151 Fab / Hemagglutinin
1QFU*	1QFU	2VIU	Igg1–k Fab / Hemagglutinin

Table 6.2: Antibody/antigen Complexes

Complex	Receptor	Ligand	Description
1IGC*	1IGC	1IGD	IgG1 Fab Fragment / Protein G
2PCC	1CCA	1YCC	Cytochrome C Peroxidase / Iso–1–Cytochrome C
1WQ1	1WER	5P21	RAS activating domain/ RAS
1AVZ	1AVV	1SHF	HIV–1 NEF / FYN tyrosin kinase SH3 domain
1ATN*	1ATN	3DNI	Actin / Deoxyribonuclease I
1GLA*	1GLA	1F3G	Glycerol kinase / GSF III
1SPB*	1SUP	1SPB	Subtilisin / Subtilisin prosegment
2BTF*	2BTF	1PNE	beta–Actin / Profilin
1MDA	2BBK	1AAN	Methylamine dehydrogenase / Amicyanin
1A0O*	1CHN	1A0O	Che A / Che Y

Table 6.3: Other Complexes

Complex	Receptor	Ligand	Description
1BTH	2HNT	6PTI	Thrombin mutant/ Pancreatic trypsin inhibitor
1IAI*	1AIF	1IAI	IgG1 Idiotypic Fab / Igg2A Anti–Idiotypic Fab
1FIN	1HCL	1VIN	CDK2 cyclin–dependant kinase 2 / Cyclin
3HHR*	3HHR	1HGU	Human growth hormone / Receptor
1FQ1	1B39	1FPZ	CDK2 / KAP
1GOT	1TAG	1TBG	transducin Gt–alpha, Gi–alpha chimera / Gt–beta–gamma
1EFU*	1D8T	1EFU	E. coli EFtu / EFts

Table 6.4: Difficult Test Cases

6.3 C: Sourcecode of work done

complex.hh

```
// ************************************************
// File: complex.hh
// purpose: Complex math functionality
//
// author: Hans-Henrik Staerfeldt
// notes:

#include <iostream.h>
#include <fftw.h>

typedef fftw_complex Complex;

Complex operator*(const Complex c1, const Complex c2);
Complex operator*(const Complex c1, const double d);
Complex operator+(const Complex c1, const Complex c2);
Complex operator-(const Complex c1, const Complex c2);
Complex mulamp(const Complex c1, const Complex c2);
Complex minamp(const Complex c1, const Complex c2);
Complex maxamp(const Complex c1, const Complex c2);
double amplitude(const Complex &c);
Complex conjugate(const Complex &c);
ostream &operator<<(ostream &s,const Complex &c);
```

complex.cc

```
// ************************************************
// File: complex.cc
// purpose: Complex math functionality
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "complex.hh"
#include <math.h>

Complex operator*(const Complex c1, const double d) {
   Complex cr;
   cr.re=c1.re*d;
   cr.im=c1.im*d;
   return cr;
}

Complex operator*(const Complex c1, const Complex c2) {
   Complex cr;
   cr.re=c1.re*c2.re-c1.im*c2.im;
   cr.im=c1.re*c2.im+c1.im*c2.re;
   return cr;
}

Complex operator+(const Complex c1, const Complex c2) {
   Complex cr;
   cr.re=c1.re+c2.re;
   cr.im=c1.im+c2.im;
   return cr;
}

Complex operator-(const Complex c1, const Complex c2) {
   Complex cr;
   cr.re=c1.re-c2.re;
   cr.im=c1.im-c2.im;
   return cr;
}

Complex mulamp(const Complex c1, const Complex c2) {
   Complex cr;
   cr.re=c1.re*c2.re;
   cr.im=c1.im*c2.im;
   return cr;
}

Complex minamp(const Complex c1, const Complex c2) {
   Complex cr;

   if (c1.im>c2.im)
     cr.im=c2.im;
   else
     cr.im=c1.im;

   if (c1.re>c2.re)
     cr.re=c2.re;
   else
     cr.re=c1.re;

   return cr;
}

Complex maxamp(const Complex c1, const Complex c2) {
   Complex cr;

   if (c1.im<c2.im)
     cr.im=c2.im;
   else
     cr.im=c1.im;

   if (c1.re<c2.re)
     cr.re=c2.re;
   else
     cr.re=c1.re;

   return cr;
}

double amplitude(const Complex &c) {
   return sqrt(c.re*c.re+c.im*c.im);
}

Complex conjugate(const Complex &c) {
   Complex r;
   r.re=c.re;r.im=-c.im;
   return r;
}

ostream &operator<<(ostream &s,const Complex &c) {
   s<<"{"<<c.re<<","<<c.im<<"}";
   return s;
}
```

vector3d.hh

```
// ************************************************
// File: vector3d.hh
// purpose: 3d and 4d vector and matrix manipulation
// quaternion math
// author: Hans-Henrik Staerfeldt
// notes:

#ifndef VECTOR3D_HH
#define VECTOR3D_HH
#include <math.h>
#include <stream.h>


#define PI 3.14159265358979323846
#define PI2 6.28318530717958647692
#define radiantodegree(x) ((x) * 180.0 /PI)
#define degreetoradian(x) ((x) * PI /180.0)

class Vector4D;
class Matrix4D;
class Vector3D;
class Matrix3D;
class Quaternion;


// ************************************************
// class: Vector3D
// purpose: 3d vectors

class Vector3D {
friend Vector3D operator+(double d, const Vector3D &v1);
friend Vector3D operator+(const Vector3D &v1, double d);
friend Vector3D operator+(const Vector3D &v1,const Vector3D &v2);
friend Vector3D operator-(double d, const Vector3D &v1);
friend Vector3D operator-(const Vector3D &v1, double d);
friend Vector3D operator-(const Vector3D &v1,const Vector3D &v2);
friend Vector3D operator*(double d, const Vector3D &v1);
friend Vector3D operator*(const Vector3D &v1, double d);

friend Vector3D operator*(const Vector3D &v1,const Vector3D &v2);
friend double operator%(const Vector3D &v1,const Vector3D &v2);

friend Vector3D operator/(const Vector3D &v1, double d);
friend Vector3D operator*(const Vector3D &v1, const Matrix3D &m1);
friend Vector3D operator*(const Vector3D &v1, const Matrix4D &m1);
friend Vector3D operator*(const Matrix3D &m1, const Vector3D &v1);
friend Vector3D operator*(const Matrix4D &m1, const Vector3D &v1);
friend ostream &operator<<(ostream &s,const Vector3D &v1);

friend class Vector4D;
friend class Matrix3D;
friend class Matrix4D;
friend class Quaternion;

 public:
   Vector3D(const double x,const double y,const double z){set(x,y,z);};
```

```
   Vector3D(){clear();};

   inline double length()
   {return sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);};
   inline void set(const double x,const double y,const double z)
   {v[0]=x;v[1]=y;v[2]=z;};

   void setspherical(const double phi,const double theta,const double r);
   void getspherical(double &phi, double &theta, double &r);

   static double distance(const Vector3D &v1, const Vector3D &v2);
   static double distancesquared(const Vector3D &v1, const Vector3D &v2);
   static double dihedral(const Vector3D &v1,const Vector3D &v2,
                          const Vector3D &v3,const Vector3D &v4);
   void normalize();
   inline void clear(){set(0.0,0.0,0.0);};

   inline double &operator[](const int idx) { return v[idx]; };
   inline double operator[](const int idx) const { return v[idx]; };
   inline double &x(){ return v[0]; };
   inline double x() const{ return v[0]; };
   inline double &y(){ return v[1]; };
   inline double y() const{ return v[1]; };
   inline double &z(){ return v[2]; };
   inline double z() const{ return v[2]; };

   Vector3D &operator=(const Vector3D &v1);

   inline Vector3D operator-() const
   {return Vector3D(-v[0],-v[1],-v[2]);};

   inline Vector3D &operator-=(const double d)
   {set(v[0]-d,v[1]-d,v[2]-d);return *this;};
   inline Vector3D &operator+=(const double d)
   {set(v[0]+d,v[1]+d,v[2]+d);return *this;};
   inline Vector3D &operator* =(const double d)
   {set(v[0]*d,v[1]*d,v[2]*d);return *this;};
   inline Vector3D &operator/=(const double d)
   {set(v[0]/d,v[1]/d,v[2]/d);return *this;};

   inline Vector3D &operator-=(const Vector3D &v2)
   {set(v[0]-v2.v[0],v[1]-v2.v[1],v[2]-v2.v[2]);return *this;};
   inline Vector3D &operator+=(const Vector3D &v2)
   {set(v[0]+v2.v[0],v[1]+v2.v[1],v[2]+v2.v[2]);return *this;};

   Vector3D &operator* =(const Matrix3D &m1);
   Vector3D &operator* =(const Matrix4D &m1);

 private:
   double v[3];
};

inline Vector3D operator+(double d, const Vector3D &v1) {
   return Vector3D(v1.v[0]+d,v1.v[1]+d,v1.v[2]+d);
}
inline Vector3D operator+(const Vector3D &v1, double d) {
   return Vector3D(v1.v[0]+d,v1.v[1]+d,v1.v[2]+d);
}
inline Vector3D operator+(const Vector3D &v1,const Vector3D &v2) {
   return Vector3D(v1.v[0]+v2.v[0],v1.v[1]+v2.v[1],v1.v[2]+v2.v[2]);
}
inline Vector3D operator-(double d, const Vector3D &v1){
   return Vector3D(v1.v[0]-d,v1.v[1]-d,v1.v[2]-d);
}
inline Vector3D operator-(const Vector3D &v1, double d) {
   return Vector3D(v1.v[0]-d,v1.v[1]-d,v1.v[2]-d);
}
inline Vector3D operator-(const Vector3D &v1,const Vector3D &v2) {
   return Vector3D(v1.v[0]-v2.v[0],v1.v[1]-v2.v[1],v1.v[2]-v2.v[2]);
}
inline Vector3D operator*(double d, const Vector3D &v1) {
   return Vector3D(v1.v[0]*d,v1.v[1]*d,v1.v[2]*d);
}
inline Vector3D operator*(const Vector3D &v1, double d) {
   return Vector3D(v1.v[0]*d,v1.v[1]*d,v1.v[2]*d);
}
inline Vector3D operator/(const Vector3D &v1, double d) {
   return Vector3D(v1.v[0]/d,v1.v[1]/d,v1.v[2]/d);
}
inline double operator%(const Vector3D &v1,const Vector3D &v2){
   return v1.v[0]*v2.v[0]+v1.v[1]*v2.v[1]+v1.v[2]*v2.v[2];
}



// ***********************************************
// class: Vector4D
// purpose: 4d vector manipulation
// notes:

class Vector4D {
friend Vector4D operator+(double d, const Vector4D &v1);
friend Vector4D operator+(const Vector4D &v1, double d);
friend Vector4D operator+(const Vector4D &v1,const Vector4D &v2);
friend Vector4D operator-(double d, const Vector4D &v1);
friend Vector4D operator-(const Vector4D &v1, double d);
friend Vector4D operator-(const Vector4D &v1,const Vector4D &v2);
friend Vector4D operator*(double d, const Vector4D &v1);
friend Vector4D operator*(const Vector4D &v1, double d);
friend double operator%(const Vector4D &v1,const Vector4D &v2);
friend Vector4D operator/(const Vector4D &v1, double d);
friend Vector4D operator*(const Vector4D &v1, const Matrix4D &m1);
friend Vector4D operator*(const Matrix4D &m1, const Vector4D &v1);
friend ostream &operator≪(ostream &s,const Vector4D &v1);

friend class Vector3D;
friend class Matrix4D;
friend class Quaternion;

 public:
   Vector4D(const double x,const double y,const double z,const double w)
   {set(x,y,z,w);};
   Vector4D(){clear();};
   double length();
   inline void set(const double x,const double y,const double z,const double w)
   {v[0]=x;v[1]=y;v[2]=z;v[3]=w;};

   static double distance(const Vector4D &v1, const Vector4D &v2);
   static double distancesquared(const Vector4D &v1, const Vector4D &v2);
   void normalize();
   void clear() {set(0.0,0.0,0.0,0.0);};

   inline double &operator[](const int idx) { return v[idx]; };
   inline double operator[](const int idx) const { return v[idx]; };
   inline double &x(){ return v[0]; };
   inline double x() const{ return v[0]; };
   inline double &y(){ return v[1]; };
   inline double y() const{ return v[1]; };
   inline double &z(){ return v[2]; };
   inline double z() const{ return v[2]; };
   inline double &w(){ return v[3]; };
   inline double w() const{ return v[3]; };

   Vector4D &operator=(const Vector4D &v1);

   inline Vector4D operator-() const
   {return Vector4D(-v[0],-v[1],-v[2],-v[3]);};
   inline Vector4D &operator-=(const double d)
   {set(v[0]-d,v[1]-d,v[2]-d,v[3]-d);return *this;};
   inline Vector4D &operator+=(const double d)
   {set(v[0]+d,v[1]+d,v[2]+d,v[3]+d);return *this;};
   inline Vector4D &operator* =(const double d)
   {set(v[0]*d,v[1]*d,v[2]*d,v[3]*d);return *this;};
   inline Vector4D &operator/=(const double d)
   {set(v[0]/d,v[1]/d,v[2]/d,v[3]/d);return *this;};
   inline Vector4D &operator-=(const Vector4D &v2)
   {set(v[0]-v2.v[0],v[1]-v2.v[1],v[2]-v2.v[2],v[3]-v2.v[3]);return *this;};
   inline Vector4D operator+=(const Vector4D &v2)
   {set(v[0]+v2.v[0],v[1]+v2.v[1],v[2]+v2.v[2],v[3]+v2.v[3]);return *this;};

   Vector4D &operator* =(const Matrix4D &m1);

 private:
   double v[4];
};

inline Vector4D operator+(double d, const Vector4D &v1){
   return Vector4D(v1.v[0]+d,v1.v[1]+d,v1.v[2]+d,v1.v[3]+d);
}
inline Vector4D operator+(const Vector4D &v1, double d){
   return Vector4D(v1.v[0]+d,v1.v[1]+d,v1.v[2]+d,v1.v[3]+d);
}
inline Vector4D operator+(const Vector4D &v1,const Vector4D &v2){
   return Vector4D(v1.v[0]+v2.v[0],v1.v[1]+v2.v[1],
                   v1.v[2]+v2.v[2],v1.v[3]+v2.v[3]);
}
inline Vector4D operator-(double d, const Vector4D &v1){
   return Vector4D(v1.v[0]-d,v1.v[1]-d,v1.v[2]-d,v1.v[3]-d);
}
inline Vector4D operator-(const Vector4D &v1, double d){
   return Vector4D(v1.v[0]-d,v1.v[1]-d,v1.v[2]-d,v1.v[3]-d);
}
inline Vector4D operator-(const Vector4D &v1,const Vector4D &v2) {
   return Vector4D(v1.v[0]-v2.v[0],v1.v[1]-v2.v[1],
                   v1.v[2]-v2.v[2],v1.v[3]-v2.v[3]);
}
inline Vector4D operator*(double d, const Vector4D &v1){
   return Vector4D(v1.v[0]*d,v1.v[1]*d,v1.v[2]*d,v1.v[3]*d);
}
inline Vector4D operator*(const Vector4D &v1, double d){
```

```
    return Vector4D(v1.v[0]*d,v1.v[1]*d,v1.v[2]*d,v1.v[3]*d);
}
inline double operator%(const Vector4D &v1,const Vector4D &v2){
    return v1.v[0]*v2.v[0]+v1.v[1]*v2.v[1]+v1.v[2]*v2.v[2]+v1.v[3]*v2.v[3];
}
inline Vector4D operator/(const Vector4D &v1, double d){
    return Vector4D(v1.v[0]/d,v1.v[1]/d,v1.v[2]/d,v1.v[3]/d);
}
double distance(const Vector4D &v1, const Vector4D &v2);
double distancesquared(const Vector4D &v1, const Vector4D &v2);


// ************************************************
// class: Matrix3D
// purpose: 3d matrix manipulation
// notes:

class Matrix3D {
friend Vector3D operator*(const Vector3D &v1, const Matrix3D &m1);
friend Vector3D operator*(const Matrix3D &m1, const Vector3D &v1);
friend ostream &operator<<(ostream &s,const Matrix3D &m1);

friend class Vector3D;
friend class Vector4D;

 public:
    Matrix3D(){clear();};
    Matrix3D(const Vector3D &c1,const Vector3D &c2,const Vector3D &c3);
    Matrix3D(const double x11, const double x21, const double x31,
             const double x12, const double x22, const double x32,
             const double x13, const double x23, const double x33);
    void clear();
    void set(const double x11, const double x21, const double x31,
             const double x12, const double x22, const double x32,
             const double x13, const double x23, const double x33);
    void flip();

    void rotation(const double psi, const double theta, const double phi);
    void rotation(const Quaternion &q);

    inline double &operator()(const int i, const int j) {return m[i][j];};
    inline double operator()(const int i, const int j) const {return m[i][j];};

    Matrix3D &operator=(const Matrix3D &m1);

 private:
    double m[3][3];
};

Vector3D operator*(const Vector3D &v1, const Matrix3D &m1);
Vector3D operator*(const Matrix3D &m1, const Vector3D &v1);
ostream &operator<<(ostream &s,const Matrix3D &m1);

// ************************************************
// class: Matrix4D
// purpose: 4d matrix manipulation
// notes:

class Matrix4D {
friend Vector3D operator*(const Vector3D &v1, const Matrix4D &m1);
friend Vector3D operator*(const Matrix4D &m1, const Vector3D &v1);
friend ostream &operator<<(ostream &s,const Matrix4D &m1);
friend class Matrix3D;
friend class Vector3D;
friend class Vector4D;

 public:
    Matrix4D(){clear();};
    Matrix4D(const Vector4D &c1,const Vector4D &c2,
             const Vector4D &c3,const Vector4D &c4);
    void clear();
    void set(const double x11, const double x21, const double x31, const double
x41,
             const double x12, const double x22, const double x32, const
double x42,
             const double x13, const double x23, const double x33, const
double x43,
             const double x14, const double x24, const double x34, const
double x44);

    void rotation(const double psi, const double theta, const double phi);
    void rotation(const Quaternion &q);
    void viewat(const Vector3D &v1, const Vector3D &v2, const Vector3D &v3);
    void iviewat(const Vector3D &v1, const Vector3D &v2, const Vector3D &v3);
    void flip();

    double &operator()(const int i, const int j) { return m[i][j]; };
    double operator()(const int i, const int j) const { return m[i][j]; };

    Matrix4D &operator=(const Matrix4D &m1);

 private:
    double m[4][4];
};


// ************************************************
// class: Quaternion
// purpose: quaternion math
// notes: usefull for rotations

class Quaternion {
friend Quaternion operator*(const Quaternion &q1,const Quaternion &q2);
friend ostream &operator<<(ostream &s,const Quaternion &q1);
friend class Matrix4D;

 public:
    Quaternion(){clear();};
    Quaternion(const double w,const double x,const double y,const double z)
    {set(w,x,y,z);};
    Quaternion(const Vector3D &v1,const double theta);

    double length();
    inline void set(const double w,const double x,const double y,const double z)
    {v[0]=w;v[1]=x;v[2]=y;v[3]=z;};
    void set(const Vector3D &v1, const double theta);
    void normalize();
    void clear() {set(1.0,0.0,0.0,0.0);};

    inline double &operator[](const int idx) { return v[idx]; };
    inline double operator[](const int idx) const { return v[idx]; };
    inline double &w(){ return v[0]; };
    inline double w() const{ return v[0]; };
    inline double &x(){ return v[1]; };
    inline double x() const{ return v[1]; };
    inline double &y(){ return v[2]; };
    inline double y() const{ return v[2]; };
    inline double &z(){ return v[3]; };
    inline double z() const{ return v[3]; };

    Quaternion &operator=(const Quaternion &q1);

 private:
    double v[4];
};

Quaternion operator*(const Quaternion &q1,const Quaternion &q2);

ostream &operator<<(ostream &s,const Vector4D &v1);
ostream &operator<<(ostream &s,const Vector3D &v1);
ostream &operator<<(ostream &s,const Quaternion &q1);
ostream &operator<<(ostream &s,const Matrix4D &m1);
ostream &operator<<(ostream &s,const Matrix3D &m1);

#endif
```

vector3d.cc

```
// ************************************************
// File: vector3d.cc
// purpose: 3d and 4d vector and matrix manipulation
// quaternion math
// author: Hans-Henrik Staerfeldt
// notes:

#include <stddef.h>
#include <string.h>
#include <assert.h>
#include "vector3d.hh"


// ************************************************
// method: Vector3D::dihedral
// purpose: get dihedral angles from 3 vectors.
// notes:

double Vector3D::dihedral(const Vector3D &v1,
                          const Vector3D &v2,
                          const Vector3D &v3,
                          const Vector3D &v4) {
    double magN1,magN2,cos_theta;
    double result;
    Vector3D A,B,C,N1,N2,N;

    /* Define three vectors */
    A=v2-v1;
    B=v2-v3;
    C=v3-v4;
```

```
    N1=A*B;
    N2=C*B;
    N=N2*N1;

    /* angle between N1 and N2 is the dihedral */
    magN1 = N1.length();
    magN2 = N2.length();

    cos_theta = N1%N2/(magN1*magN2);

    if (cos_theta < -1.0) cos_theta = -1;
    if (cos_theta > 1.0) cos_theta = 1;

    result = radiantodegree(acos(cos_theta));
    if (B%N < 0) result = -result;

    return result;
}

// **************************************************
// method: Vector3D::distance
// purpose: get euclidian distance between two 3d vectors
// notes:

double Vector3D::distance(const Vector3D &v1, const Vector3D &v2){
    Vector3D r;
    r=v1-v2;
    return r.length();
}

// **************************************************
// method: Vector3D::distancesquared
// purpose: get squared distance between two v3d vectors
// notes: this might speed up some calculations

double Vector3D::distancesquared(const Vector3D &v1, const Vector3D &v2){
    Vector3D r;
    r=v1-v2;
    return r[0]*r[0]+r[1]*r[1]+r[2]*r[2];
}

// **************************************************
// method: Vector3D::setspherical
// purpose: set vector from spherical coordinates
// notes:
// Using this convention; theta is the angle out of the zaxis,
// the angle going between 0 and PI degrees only.
// phi is the rotation around the z axis.

void Vector3D::setspherical(const double phi,
                            const double theta,
                            const double r){
    v[0]=r*sin(theta)*cos(phi);
    v[1]=r*sin(theta)*sin(phi);
    v[2]=r*cos(theta);
}

// **************************************************
// method: Vector3D::getspherical
// purpose: get spherical coordinates from vector
// notes: see notes on Vector3D::setspherical

void Vector3D::getspherical(double &phi, double &theta, double &r) {
    double rxy=sqrt(x()*x()+y()*y());
    r=length();

    if (x()>0 && y()>0) {
        phi=asin(y()/rxy); // 1
    }else if (x()<=0 && y()>0) {
        phi=acos(x()/rxy); // 2
    }else if (x()<=0 && y()<=0) {
        phi=asin(-y()/rxy)-PI; // 3
    }else if (x()>0 && y()<=0) {
        phi=asin(y()/rxy); // 4
    }

    theta=acos(z()/r);
    if (theta<0) theta=-theta;
}

// **************************************************
// method: Vector3D::normalize
// purpose: normalize vector to get length 1.0
// notes:

void Vector3D::normalize() {
    double len=length();
    if (len<0.0000001) throw "vector too short";
    v[0]/=len;
    v[1]/=len;
    v[2]/=len;
}

ostream &operator<<(ostream &s,const Vector3D &v1) {
    s << "Vector3D("<<v1[0]<<","<<v1[1]<<","<<v1[2]<<")";
    return s;
}

Vector3D &Vector3D::operator=(const Vector3D &v1) {
    memcpy(v,v1.v,3*sizeof(double));
    return *this;
}


// **************************************************
// method: operator*(const Vector3D &v1,const Vector3D &v2)
// purpose: cross product of 2 3d vectors
// notes:

Vector3D operator*(const Vector3D &v1,const Vector3D &v2){
    // cross product
    return Vector3D(v1.v[1]*v2.v[2] - v2.v[1]*v1.v[2],
                    v1.v[2]*v2.v[0] - v2.v[2]*v1.v[0],
                    v1.v[0]*v2.v[1] - v2.v[0]*v1.v[1]);
}

// **************************************************
// method: Vector3D*Matrix3D
// purpose: vector and matrix multiplication
// notes:

Vector3D operator*(const Vector3D &v1, const Matrix3D &m1){
    Vector3D r;
    int i,j;
    for (i=0;i<3;i++) {
        r.v[i]=0.0;
        for (j=0;j<3;j++) {
            r.v[i]+=m1.m[i][j]*v1.v[j];
        }
    }
    return r;
}

Vector3D operator*(const Matrix3D &m1, const Vector3D &v1){
    Vector3D r;
    int i,j;
    for (i=0;i<3;i++) {
        r.v[i]=0.0;
        for (j=0;j<3;j++) {
            r.v[i]+=m1.m[i][j]*v1.v[j];
        }
    }
    return r;
}

// **************************************************
// method: Vector3D*Matrix4D
// purpose: vector and matrix multiplication
// notes:

Vector3D operator*(const Vector3D &v1, const Matrix4D &m1) {
    Vector4D o(v1.v[0],v1.v[1],v1.v[2],1.0);
    Vector4D r;

    r=m1*o;
    return Vector3D(r[0],r[1],r[2]);
}

Vector3D operator*(const Matrix4D &m1, const Vector3D &v1) {
    Vector4D o(v1.v[0],v1.v[1],v1.v[2],1.0);
    Vector4D r;

    r=m1*o;
    return Vector3D(r[0],r[1],r[2]);
}

Vector3D &Vector3D::operator* =(const Matrix3D &m1){
    Vector3D o(v[0],v[1],v[2]);
    int i,j;
    for (i=0;i<3;i++) {
        v[i]=0.0;
        for (j=0;j<3;j++) {
            v[i]+=m1.m[i][j]*o.v[j];
        }
    }
    return *this;
}

Vector3D &Vector3D::operator* =(const Matrix4D &m1){
    Vector4D o(v[0],v[1],v[2],1.0);
```

```
    Vector4D r;
    int i,j;

    for (i=0;i<4;i++) {
        r.v[i]=0.0;
        for (j=0;j<4;j++) {
            r.v[i]+=m1.m[i][j]*o.v[j];
        }
    }
    set(r.v[0],r.v[1],r.v[2]);
    return *this;
}


// ***************************************************
// method: Vector4D::length
// purpose: length of 4d vector
// notes:

double Vector4D::length() {
    return sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]+v[3]*v[3]);
}

// ***************************************************
// method: Vector4D::distance
// purpose: euclidian distance between 4d vectors
// notes:

double Vector4D::distance(const Vector4D &v1, const Vector4D &v2){
    Vector4D r;
    r=v1-v2;
    return r.length();
}

// ***************************************************
// method: Vector4D::distancesquared
// purpose: squared distance between two 4d vectors
// notes:

double Vector4D::distancesquared(const Vector4D &v1, const Vector4D &v2){
    Vector4D r;
    r=v1-v2;
    return r[0]*r[0]+r[1]*r[1]+r[2]*r[2]+r[3]*r[3];
}

// ***************************************************
// method: Vector4D::normalize
// purpose: normalize 4d vector
// notes:

void Vector4D::normalize(void) {
    double len=length();
    if (len<0.0000001) throw "vector too short";
    v[0]/=len;
    v[1]/=len;
    v[2]/=len;
    v[3]/=len;
}

ostream &operator<<(ostream &s,const Vector4D &v1) {
    s << "Vector4D("<<v1.v[0]<<","<<v1.v[1];
    s <<","<<v1.v[2]<<","<<v1.v[3]<<")";
    return s;
}

Vector4D &Vector4D::operator=(const Vector4D &v1) {
    memcpy(v,v1.v,4*sizeof(double));
    return *this;
}

// ***************************************************
// method: Vector4D*Matrix4D
// purpose: 4d vector matrix multiplication
// notes:

Vector4D &Vector4D::operator*=(const Matrix4D &m1){
    Vector4D o(v[0],v[1],v[2],v[3]);
    int i,j;

    for (i=0;i<4;i++) {
        v[i]=0.0;
        for (j=0;j<4;j++) {
            v[i]+=m1(i,j)*o.v[j];
        }
    }
    return *this;
}

Vector4D operator*(const Vector4D &v1, const Matrix4D &m1){
    Vector4D r;
    int i,j;

    for (i=0;i<4;i++) {
        r.v[i]=0.0;
        for (j=0;j<4;j++) {
            r.v[i]+=m1(i,j)*v1.v[j];
        }
    }
    return r;
}

Vector4D operator*(const Matrix4D &m1, const Vector4D &v1){
    Vector4D r;
    int i,j;

    for (i=0;i<4;i++) {
        r.v[i]=0.0;
        for (j=0;j<4;j++) {
            r.v[i]+=m1(i,j)*v1.v[j];
        }
    }
    return r;
}


Matrix3D::Matrix3D(const Vector3D &c1,
                   const Vector3D &c2,
                   const Vector3D &c3) {
    set(c1[0], c2[0], c3[0],
        c1[1], c2[1], c3[1],
        c1[2], c2[2], c3[2]);
}

Matrix3D::Matrix3D(const double x11, const double x21, const double x31,
                   const double x12, const double x22, const double x32,
                   const double x13, const double x23, const double x33) {
    set (x11,x21,x31,
         x12,x22,x32,
         x13,x23,x33);
}

void Matrix3D::clear() {
    set(1.0, 0.0, 0.0,
        0.0, 1.0, 0.0,
        0.0, 0.0, 1.0);
}

void Matrix3D::set(const double x11, const double x21, const double x31,
                   const double x12, const double x22, const double x32,
                   const double x13, const double x23, const double x33) {
    m[0][0]=x11; m[1][0]=x21; m[2][0]=x31;
    m[0][1]=x12; m[1][1]=x22; m[2][1]=x32;
    m[0][2]=x13; m[1][2]=x23; m[2][2]=x33;
}

void Matrix3D::flip() {
    set (m[0][0],m[0][1],m[0][2],
         m[1][0],m[1][1],m[1][2],
         m[2][0],m[2][1],m[2][2]);
}

// ***************************************************
// method: Matrix3D::rotation
// purpose: build rotation from eulerian angles
// notes:

void Matrix3D::rotation(const double psi,
                        const double theta,
                        const double phi) {
    double A,C,E,AD;
    double B,D,F,BD;

    A=cos(psi);
    B=sin(psi);
    C=cos(theta);
    D=sin(theta);
    E=cos(phi);
    F=sin(phi);
    AD=A*D;
    BD=B*D;

    set( C * E, -C * F, -D,
        -BD * E + A * F, BD * F + A * E, -B * C,
         AD * E + B * F, -AD * F + B * E, A * C);
}

// ***************************************************
// method: Matrix3D::rotation
// purpose: build rotation from quaternion
// notes: 4d rotation around a vector
```

```
void Matrix3D::rotation(const Quaternion &q) {

    register double wx, wy, wz, xx, yy, yz, xy, xz, zz, x2, y2, z2;

    x2 = q[1] + q[1]; y2 = q[2] + q[2]; z2 = q[3] + q[3];
    wx = q[0] * x2; wy = q[0] * y2; wz = q[0] * z2;
    xx = q[1] * x2; xy = q[1] * y2; xz = q[1] * z2;
    yy = q[2] * y2; yz = q[2] * z2; zz = q[3] * z2;

    set(1.0 - (yy + zz), xy - wz, xz + wy,
        xy + wz, 1.0 - (xx + zz), yz - wx,
        xz - wy, yz + wx, 1.0 - (xx + yy));
}

ostream &operator<<(ostream &s,const Matrix3D &m1) {
    s << "Matrix3D("<<m1.m[0][0]<<","<<m1.m[1][0];
    s <<","<<m1.m[2][0]<<endl;
    s << " "<<m1.m[0][1]<<","<<m1.m[1][1];
    s <<","<<m1.m[2][1]<<endl;
    s << " "<<m1.m[0][2]<<","<<m1.m[1][2];
    s <<","<<m1.m[2][2]<<")"<<endl;
    return s;
}

Matrix3D &Matrix3D::operator=(const Matrix3D &m1) {
    memcpy(m,m1.m,3*3*sizeof(double));
    return *this;
};


Matrix4D::Matrix4D(const Vector4D &c1,const Vector4D &c2,
                   const Vector4D &c3,const Vector4D &c4){
    set(c1[0],c2[0],c3[0],c4[0],
        c1[1],c2[1],c3[1],c4[1],
        c1[2],c2[2],c3[2],c4[2],
        c1[3],c2[3],c3[3],c4[3]);
}

void Matrix4D::clear() {
    set(1.0,0.0,0.0,0.0,
        0.0,1.0,0.0,0.0,
        0.0,0.0,1.0,0.0,
        0.0,0.0,0.0,1.0);
}

void Matrix4D::set(const double x11, const double x21, const double x31, const
double x41,
                   const double x12, const double x22, const double x32,
const double x42,
                   const double x13, const double x23, const double x33,
const double x43,
                   const double x14, const double x24, const double x34,
const double x44){

    m[0][0]=x11; m[1][0]=x21; m[2][0]=x31; m[3][0]=x41;
    m[0][1]=x12; m[1][1]=x22; m[2][1]=x32; m[3][1]=x42;
    m[0][2]=x13; m[1][2]=x23; m[2][2]=x33; m[3][2]=x43;
    m[0][3]=x14; m[1][3]=x24; m[2][3]=x34; m[3][3]=x44;
}

void Matrix4D::flip() {
    set (m[0][0],m[0][1],m[0][2],m[0][3],
         m[1][0],m[1][1],m[1][2],m[1][3],
         m[2][0],m[2][1],m[2][2],m[2][3],
         m[3][0],m[3][1],m[3][2],m[3][3]);
}

// ************************************************
// method: Matrix4D::viewat
// purpose: create coordinate system change matrix
// notes:
// transformation of coordinatsystem looking from v1
// toward v2, with v3 as up.
void Matrix4D::viewat(const Vector3D &v1,
                      const Vector3D &v2,
                      const Vector3D &v3) {
    Vector3D v12;
    Vector3D v13;
    Vector3D t1;
    Vector3D t2;

    v12=v2-v1;
    v13=v3-v1;

    try {
        t1=v12;t1.normalize();
        m[2][0] = t1[0];
        m[2][1] = t1[1];
        m[2][2] = t1[2];

        t2=v13*v12;
        t2.normalize();
        m[0][0] = t2[0];
        m[0][1] = t2[1];
        m[0][2] = t2[2];

        m[3][0] = 0.;
        m[3][1] = 0.;
        m[3][2] = 0.;

        t2=t1*t2;
        t2.normalize();
        m[1][0] = t2[0];
        m[1][1] = t2[1];
        m[1][2] = t2[2];

        m[0][3] = -v1[0]*m[0][0] - v1[1]*m[0][1] - v1[2]*m[0][2];
        m[1][3] = -v1[0]*m[1][0] - v1[1]*m[1][1] - v1[2]*m[1][2];
        m[2][3] = -v1[0]*m[2][0] - v1[1]*m[2][1] - v1[2]*m[2][2];
        m[3][3] = 1.0;
    }
    catch (char *str) {
        throw "unable to make matrix, invalid point set";
    }
}

// ************************************************
// method: Matrix4D::viewat
// purpose: create inverse coordinate system change matrix
// notes:
// transformation of coordinatsystem looking from v1
// toward v2, with v3 as up.
void Matrix4D::iviewat(const Vector3D &v1,
                       const Vector3D &v2,
                       const Vector3D &v3) {
    Vector3D v12;
    Vector3D v13;
    Vector3D t1;
    Vector3D t2;

    v12=v2-v1;
    v13=v3-v1;

    try {
        t1=v12;t1.normalize();
        m[0][2] = t1[0];
        m[1][2] = t1[1];
        m[2][2] = t1[2];

        t2=v13*v12;
        t2.normalize();
        m[0][0] = t2[0];
        m[1][0] = t2[1];
        m[2][0] = t2[2];

        m[3][0] = 0.0;
        m[3][1] = 0.0;
        m[3][2] = 0.0;

        t2=t1*t2;
        t2.normalize();
        m[0][1] = t2[0];
        m[1][1] = t2[1];
        m[2][1] = t2[2];

        //translation
        m[0][3] = v1[0];
        m[1][3] = v1[1];
        m[2][3] = v1[2];
        m[3][3] = 1.0;
    }
    catch (char *str) {
        throw "unable to make matrix, invalid point set";
    }
}

// ************************************************
// method: Matrix4D::rotation
// purpose: create rotation from eulerian angles
// notes:

void Matrix4D::rotation(const double psi,
                        const double theta,
                        const double phi) {
    double A,C,E,AD;
    double B,D,F,BD;

    A=cos(psi);
    B=sin(psi);
```

```
    C=cos(theta);
    D=sin(theta);
    E=cos(phi);
    F=sin(phi);
    AD=A*D;
    BD=B*D;

    set( C * E, -C * F, -D, 0.0,
         -BD * E + A * F, BD * F + A * E, -B * C, 0.0,
          AD * E + B * F, -AD * F + B * E, A * C, 0.0,
          0.0, 0.0, 0.0, 1.0);
}

// **************************************************
// method: Matrix4D::rotation
// purpose: create rotation from quaternion
// notes:

void Matrix4D::rotation(const Quaternion &q){
    register double wx, wy, wz, xx, yy, yz, xy, xz, zz, x2, y2, z2;

    x2 = q[1] + q[1];
    y2 = q[2] + q[2];
    z2 = q[3] + q[3];

    wx = q[0] * x2;
    wy = q[0] * y2;
    wz = q[0] * z2;

    xx = q[1] * x2;
    xy = q[1] * y2;
    xz = q[1] * z2;

    yy = q[2] * y2;
    yz = q[2] * z2;
    zz = q[3] * z2;

    set(1.0 - (yy + zz), xy - wz, xz + wy, 0.0,
        xy + wz, 1.0 - (xx + zz), yz - wx, 0.0,
        xz - wy, yz + wx, 1.0 - (xx + yy), 0.0,
        0.0, 0.0, 0.0, 1.0);
}

ostream &operator<<(ostream &s,const Matrix4D &m1) {
    s << "Matrix4D("<<m1.m[0][0]<<","<<m1.m[1][0]<<",";
    s <<m1.m[2][0]<<","<<m1.m[3][0]<<endl;
    s << " "<<m1.m[0][1]<<","<<m1.m[1][1]<<",";
    s <<m1.m[2][1]<<","<<m1.m[3][1]<<endl;
    s << " "<<m1.m[0][2]<<","<<m1.m[1][2]<<",";
    s <<m1.m[2][2]<<","<<m1.m[3][2]<<endl;
    s << " "<<m1.m[0][3]<<","<<m1.m[1][3]<<",";
    s <<m1.m[2][3]<<","<<m1.m[3][3]<<")"<<endl;
    return s;
}

Matrix4D &Matrix4D::operator=(const Matrix4D &m1){
    memcpy(m,m1.m,4*4*sizeof(double));
    return *this;
}


Quaternion::Quaternion(const Vector3D &v1,const double theta){
    set(v1,theta);
}

ostream &operator<<(ostream &s,const Quaternion &q1) {
    s << "Quaternion("<<q1[0]<<","<<q1[1]<<","<<q1[2];
    s <<","<<q1[3]<<")"<<endl;
    return s;
}

// **************************************************
// method: Quaternion::length
// purpose: length of quaternion
// notes: it should normally be 1 !

double Quaternion::length() {
    return sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]+v[3]*v[3]);
}

void Quaternion::set(const Vector3D &v1, const double theta){
    double st2=sin(theta/2);
    set(cos(theta/2),v1[0]*st2,v1[1]*st2,v1[2]*st2);
    normalize();
}


void Quaternion::normalize(){
    double len=length();
    if (len<0.0000001) throw "vector too short";
    v[0]/=len;
    v[1]/=len;
    v[2]/=len;
    v[3]/=len;
}

Quaternion &Quaternion::operator=(const Quaternion &q1){
    memcpy(v,q1.v,4*sizeof(double));
    return *this;
}

// **************************************************
// method: Quaternion*Quaternion
// purpose: multiplication of quaternion
// notes:
Quaternion operator*(const Quaternion &q1,const Quaternion &q2) {
    return Quaternion(q1.v[0]*q2.v[0] - q1.v[1]*q2.v[1] -
                      q1.v[2]*q2.v[2] - q1.v[3]*q2.v[3],
                      q1.v[0]*q2.v[1] + q1.v[1]*q2.v[0] +
                      q1.v[2]*q2.v[3] - q1.v[3]*q2.v[2],
                      q1.v[0]*q2.v[2] + q1.v[2]*q2.v[0] +
                      q1.v[3]*q2.v[1] - q1.v[1]*q2.v[3],
                      q1.v[0]*q2.v[3] + q1.v[3]*q2.v[0] +
                      q1.v[1]*q2.v[2] - q1.v[2]*q2.v[1]);
}
```

field.hh

```
// **************************************************
// File: field.hh
// purpose: Various field definitions
//
// author: Hans-Henrik Staerfeldt
// notes:

#ifndef __FIELD_H__
#define __FIELD_H__

#include <vector>
using namespace std;

#include <fftw.h>

double Simplefield(const double gridscale,
                   const double radius,
                   const double dist);

double Spherefield(const double gridscale,
                   const double radius,
                   const double dist);

double Electricalfield(const double gridscale,
                       const double vdwradius,
                       const double charge,
                       const double dist);

double LJfield(const double gridscale,
               const double vdwradius,
               const double vdwwell,
               const double dist);

#endif
```

field.cc

```
// **************************************************
// File: field.cc
// purpose: implementation of various energy fields.
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "field.hh"
#include "vector3d.hh"
#include "atomic.h"

// field implementation

#define SMALL 0.000001


double Simplefield(const double gridscale,
                   const double radius,
                   const double dist) {

    if (dist<radius) return 0.0;
    else return 4.0/(dist-radius/2.0);
}
```

```
double Spherefield(const double gridscale,
                   const double radius,
                   const double dist) {

   if (dist<radius || dist<gridscale) return 1.0;
   else return 0.0;
}


double Electricalfield(const double gridscale,
                       const double vdwradius,
                       const double charge,
                       const double dist) {

   static double factor=1390.084; // converts the electrostatic
                                  //energy into kilodoule per mole
   double epsilon;

   // Electrical potential
   // E= 1/(4 pi e0) sum{i≠j}( qi qj / De rij )
   // Efield{k} = sum{i}( qi / De rik }
   // Edensity{k} = qj, if j in k.
   // Efield*Edensity ~= E

   if (dist<vdwradius) return 0;

   // approximated sigmoid dielectric function
   if (dist<6) epsilon=4.0;
   else if (dist<8) epsilon=38.0*dist-224.0;
   else epsilon=80.0;

   return factor * charge/(4*PI*epsilon*dist);
}

double LJfield(const double gridscale,
               const double vdwradiusdbl,
               const double vdwwell,
               const double dist) {
  double r6,r12;
  double r;
  double E;
  double Aij;
  double Bij;
  //static long double epsilon=4*MOL*1.7119e-21;
  // E = Aij/rij^12 - Bij/rij^6
  // Aij = e ( Ri+Rj )^12
  // Bij = 2 e ( Ri+Rj )^6
  // Ri+Rj is approximated with 'vdwradiusdbl'
  // e is vdwwell
  if (dist<SMALL)
    r=SMALL;
  else
    r=dist;

  Bij=vdwradiusdbl;
  Bij=Bij*Bij*Bij;
  Bij=Bij*Bij;

  Aij=vdwradiusdbl;
  Aij=Bij*Bij;

  Aij=vdwwell*Aij;
  Bij=vdwwell*2*Bij;

  r6=r*r*r;
  r6=r6*r6;
  r12=r6*r6;
  E= -Aij/r12 + Bij/r6;
  return E;
}
```

atomiccontactenergy.hh

```
// ************************************************
// File: atomiccontactenergy.hh
// purpose: data and definitions pertaining to
// the atomic contact energy
// author: Hans-Henrik Staerfeldt
// notes:

#include "structure.hh"

#define ACE_MAXTYPE 18

enum ACEtype {
    ACE_NONE=-1,
    ACE_0=0,
    ACE_1=1,
    ACE_2=2,
    ACE_3=3,
    ACE_4=4,
    ACE_5=5,
    ACE_6=6,
    ACE_7=7,
    ACE_8=8,
    ACE_9=9,
    ACE_10=10,
    ACE_11=11,
    ACE_12=12,
    ACE_13=13,
    ACE_14=14,
    ACE_15=15,
    ACE_16=16,
    ACE_17=17,
    ACE_MAX=18
};

extern double ACEinteraction[ACE_MAXTYPE][ACE_MAXTYPE];
extern Complex ACEfield[ACE_MAXTYPE];
extern Complex ACEdensity[ACE_MAXTYPE];

ACEtype getACEtype( Residuetype rt, ResidueAtomtype at );

Complex getACEfield(Residuetype rt1, ResidueAtomtype at1);
Complex getACEdensity(Residuetype rt1, ResidueAtomtype at1);

double ACEenergy(Residuetype rt1, ResidueAtomtype at1,
                 Residuetype rt2, ResidueAtomtype at2 );

double ACEavgenergy( Residuetype rt1, ResidueAtomtype at1 );
```

atomiccontactenergy.cc

```
#include "structure.hh"
#include "atomiccontactenergy.hh"

#define REALACE

double ACEinteraction[ACE_MAXTYPE][ACE_MAXTYPE] = {
 {-0.724, -0.903, -0.722, -0.322, -0.331, -0.603, 1.147, 0.937, 0.752, 0.502, 0.405, 0.495, -0.116, -0.092, -0.412, -0.499, -1.005, -2.060},
 {-0.903, -0.842, -0.850, -0.332, -0.365, -0.531, 1.139, 0.918, 0.852, 0.452, 0.445, 0.436, -0.044, -0.165, -0.539, -0.590, -1.123, -2.028},
 {-0.722, -0.850, -0.704, -0.371, -0.308, -0.499, 1.140, 0.846, 0.788, 0.490, 0.408, 0.418, -0.045, -0.089, -0.415, -0.478, -0.963, -2.033},
 {-0.322, -0.332, -0.371, -0.016, 0.205, -0.059, 1.414, 1.075, 1.152, 0.831, 0.758, 0.649, 0.383, 0.241, -0.073, -0.166, -0.650, -1.650},
 {-0.331, -0.365, -0.308, 0.205, 0.182, -0.009, 1.502, 1.385, 1.192, 0.912, 0.872, 0.943, 0.434, 0.392, -0.062, -0.021, -0.342, -1.212},
 {-0.603, -0.531, -0.499, -0.059, -0.009, -0.469, 1.310, 1.010, 0.859, 0.671, 0.591, 0.565, 0.122, 0.000, -0.438, -0.533, -1.034, -1.800},
 {1.147, 1.139, 1.140, 1.414, 1.502, 1.310, 3.018, 2.911, 1.157, 2.635, 1.848, 2.699, 1.587, 1.557, 1.004, 1.340, 1.252, 0.700},
 {0.937, 0.918, 0.846, 1.075, 1.385, 1.010, 2.911, 2.811, 0.989, 2.439, 1.622, 2.525, 1.361, 1.277, 0.685, 0.938, 0.814, 0.411},
 {0.752, 0.852, 0.788, 1.152, 1.192, 0.859, 1.157, 0.989, -1.978, 0.695, 1.386, 0.641, 1.002, 0.642, 0.618, 0.894, 0.792, -0.029},
 {0.502, 0.452, 0.490, 0.831, 0.912, 0.671, 2.635, 2.439, 0.695, 1.589, 1.395, 1.515, 1.022, 0.948, 0.457, 0.707, 0.586, 0.021},
 {0.405, 0.445, 0.408, 0.758, 0.872, 0.591, 1.848, 1.622, 1.386, 1.395, 1.300, 1.283, 0.931, 0.829, 0.484, 0.633, 0.389, -0.046},
 {0.495, 0.436, 0.418, 0.649, 0.943, 0.565, 2.699, 2.525, 0.641, 1.515, 1.283, 1.309, 0.921, 0.777, 0.265, 0.484, 0.274, -0.253},
 {-0.116, -0.044, -0.045, 0.383, 0.434, 0.122, 1.587, 1.361, 1.002, 1.022, 0.931, 0.921, 0.559, 0.319, 0.301, 0.212, -0.155, -0.949},
 {-0.092, -0.165, -0.089, 0.241, 0.392, 0.000, 1.557, 1.277, 0.642, 0.948, 0.829, 0.777, 0.319, -0.301, -0.159, -0.132, -0.338, -1.324},
 {-0.412, -0.539, -0.415, -0.073, -0.062, -0.438, 1.004, 0.685, 0.618, 0.457, 0.484, 0.265, 0.301, -0.159, -0.314, -0.478, -0.964, -1.410},
 {-0.499, -0.590, -0.478, -0.166, -0.021, -0.533, 1.340, 0.938, 0.894, 0.707, 0.633, 0.484, 0.212, -0.132, -0.478, -0.687, -1.240, -1.784},
 {-1.005, -1.123, -0.963, -0.650, -0.342, -1.034, 1.252, 0.814, 0.792, 0.586, 0.389, 0.274, -0.155, -0.338, -0.964, -1.240, -1.873, -2.402},
 {-2.060, -2.028, -2.033, -1.650, -1.212, -1.800, 0.700, 0.411, -0.029, 0.021, -0.046, -0.253, -0.949, -1.324, -1.410, -1.784, -2.402, -3.742}
};

#ifdef REALACE

//measure;0.678486 vs 0.749052
//correlation;0.887048 vs 0.817582
Complex ACEfield[ACE_MAXTYPE] = {
 {0.284608,0},
 {0.268847,0},
 {0.276393,0},
```

```
 {0.534227,0},
 {0.632633,0},
 {0.396841,0},
 {1.67536,0},
 {1.52189,0},
 {0.636366,0},
 {1.17573,0},
 {0.965933,0},
 {1.13051,0},
 {0.692294,0},
 {0.578143,0},
 {0.270011,0},
 {0.379935,0},
 {0.189396,0},
 {-0.304065,0}
};

Complex ACEdensity[ACE_MAXTYPE] = {
 {0.361116,0},
 {0.341118,0},
 {0.350692,0},
 {0.677831,0},
 {0.80269,0},
 {0.503516,0},
 {2.1257,0},
 {1.93098,0},
 {0.807423,0},
 {1.49177,0},
 {1.22558,0},
 {1.4344,0},
 {0.878388,0},
 {0.733552,0},
 {0.342593,0},
 {0.482065,0},
 {0.24031,0},
 {-0.385794,0}
};

#else

//measure;0.19276 vs 0.749052
//correlation;0.989231 vs 0.817582
Complex ACEfield[ACE_MAXTYPE]={
 {-1.23855,0.522625},
 {-1.28587,0.509959},
 {-1.20392,0.507698},
 {-0.924598,0.784384},
 {-0.802395,0.887797},
 {-1.12331,0.643954},
 {0.287405,2.01236},
 {0.0357928,1.85966},
 {0.514997,0.707387},
 {-0.161284,1.46319},
 {-0.133951,1.20231},
 {-0.323467,1.43057},
 {-0.572998,0.928685},
 {-0.750193,0.813703},
 {-0.90289,0.457609},
 {-1.17249,0.630147},
 {-1.65708,0.464735},
 {-2.37322,-0.0393159}
};

Complex ACEdensity[ACE_MAXTYPE]={
 {0.803352,-0.475364},
 {0.835461,-0.467197},
 {0.7809,-0.461817},
 {0.582614,-0.672922},
 {0.496628,-0.751053},
 {0.721231,-0.568248},
 {-0.278969,-1.61104},
 {-0.104538,-1.49876},
 {-0.374152,-0.548358},
 {0.0440942,-1.18745},
 {0.0371824,-0.975797},
 {0.153676,-1.16817},
 {0.341865,-0.774089},
 {0.465027,-0.689012},
 {0.582311,-0.408349},
 {0.754634,-0.559244},
 {1.08499,-0.446814},
 {1.58447,-0.0712265}
};
#endif

ACEtype getACEtype( Residuetype rt, ResidueAtomtype at ){
    switch (rt) {
      case RESIDUE_ACE:
         switch (at) {
           case RATOM_C: return ACE_2;
```

```
        case RATOM_O: return ACE_3;
        case RATOM_CH3: return ACE_5;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_ALA:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
        case RATOM_CB: return ACE_5;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_ARG:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
        case RATOM_CB: return ACE_5;
        case RATOM_CD: return ACE_11;
        case RATOM_CG: return ACE_15;
        case RATOM_CZ: return ACE_9;
        case RATOM_NE: return ACE_11;
        case RATOM_NH1: return ACE_9;
        case RATOM_NH2: return ACE_9;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_ASH:
    case RESIDUE_ASN:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
// case RATOM_AD1: return ACE_10;
// case RATOM_AD2: return ACE_10;
        case RATOM_CB: return ACE_5;
        case RATOM_CG: return ACE_10;
        case RATOM_ND2: return ACE_10;
        case RATOM_OD1: return ACE_10;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_ASP:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
        case RATOM_CB: return ACE_5;
        case RATOM_CG: return ACE_8;
        case RATOM_OD1: return ACE_8;
        case RATOM_OD2: return ACE_8;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_CYM:
    case RESIDUE_CYS:
    case RESIDUE_CYX:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
        case RATOM_CB: return ACE_5;
        case RATOM_SG: return ACE_17;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_GLH:
    case RESIDUE_GLN:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
// case RATOM_AE1: return ACE_10;
// case RATOM_AE2: return ACE_10;
// case RATOM_AK1: return ACE_10;
// case RATOM_AK2: return ACE_10;
```

```
        case RATOM_CB: return ACE_5;
        case RATOM_CD: return ACE_10;
        case RATOM_CG: return ACE_15;
        case RATOM_NE2: return ACE_10;
        case RATOM_OE1: return ACE_10;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_GLU:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
        case RATOM_CB: return ACE_5;
        case RATOM_CD: return ACE_8;
        case RATOM_CG: return ACE_15;
        case RATOM_OE1: return ACE_8;
        case RATOM_OE2: return ACE_8;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_GLY:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_HID:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
// case RATOM_AD1: return ACE_13;
// case RATOM_AD2: return ACE_13;
// case RATOM_AE1: return ACE_13;
// case RATOM_AE2: return ACE_13;
// case RATOM_AK1: return ACE_13;
// case RATOM_AK2: return ACE_13;
        case RATOM_CB: return ACE_5;
        case RATOM_CD2: return ACE_13;
        case RATOM_CE1: return ACE_13;
        case RATOM_CG: return ACE_13;
        case RATOM_ND1: return ACE_13;
        case RATOM_NE2: return ACE_13;
        default: return ACE_NONE;
      }
      break;
    case RESIDUE_HIE:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
// case RATOM_AD1: return ACE_13;
// case RATOM_AD2: return ACE_13;
// case RATOM_AE1: return ACE_13;
// case RATOM_AE2: return ACE_13;
// case RATOM_AK1: return ACE_13;
// case RATOM_AK2: return ACE_13;
        case RATOM_CB: return ACE_5;
        case RATOM_CD2: return ACE_13;
        case RATOM_CE1: return ACE_13;
        case RATOM_CG: return ACE_13;
        case RATOM_ND1: return ACE_13;
        case RATOM_NE2: return ACE_13;
      default: return ACE_NONE;
      }
      break;
    case RESIDUE_HIP:
      switch (at) {
        case RATOM_N: return ACE_0;
        case RATOM_CA: return ACE_1;
        case RATOM_C: return ACE_2;
        case RATOM_O: return ACE_3;
        case RATOM_OXT: return ACE_3;
// case RATOM_AD1: return ACE_13;
// case RATOM_AD2: return ACE_13;
// case RATOM_AE1: return ACE_13;
// case RATOM_AE2: return ACE_13;
// case RATOM_AK1: return ACE_13;
// case RATOM_AK2: return ACE_13;
```

```
      case RATOM_CB: return ACE_5;
      case RATOM_CD2: return ACE_13;
      case RATOM_CE1: return ACE_13;
      case RATOM_CG: return ACE_13;
      case RATOM_ND1: return ACE_13;
      case RATOM_NE2: return ACE_13;
      default: return ACE_NONE;
     }
     break;
   case RESIDUE_HIS:
     switch (at) {
      case RATOM_N: return ACE_0;
      case RATOM_CA: return ACE_1;
      case RATOM_C: return ACE_2;
      case RATOM_O: return ACE_3;
      case RATOM_OXT: return ACE_3;
// case RATOM_AD1: return ACE_13;
// case RATOM_AD2: return ACE_13;
// case RATOM_AE1: return ACE_13;
// case RATOM_AE2: return ACE_13;
// case RATOM_AK1: return ACE_13;
// case RATOM_AK2: return ACE_13;
      case RATOM_CB: return ACE_5;
      case RATOM_CD2: return ACE_13;
      case RATOM_CE1: return ACE_13;
      case RATOM_CG: return ACE_13;
      case RATOM_ND1: return ACE_13;
      case RATOM_NE2: return ACE_13;
      default: return ACE_NONE;
     }
     break;
   case RESIDUE_ILE:
     switch (at) {
      case RATOM_N: return ACE_0;
      case RATOM_CA: return ACE_1;
      case RATOM_C: return ACE_2;
      case RATOM_O: return ACE_3;
      case RATOM_OXT: return ACE_3;
      case RATOM_CB: return ACE_5;
      case RATOM_CD1: return ACE_16;
      case RATOM_CG1: return ACE_15;
      case RATOM_CG2: return ACE_16;
      default: return ACE_NONE;
     }
     break;
   case RESIDUE_LEU:
     switch (at) {
      case RATOM_N: return ACE_0;
      case RATOM_CA: return ACE_1;
      case RATOM_C: return ACE_2;
      case RATOM_O: return ACE_3;
      case RATOM_OXT: return ACE_3;
      case RATOM_CB: return ACE_5;
      case RATOM_CD1: return ACE_16;
      case RATOM_CD2: return ACE_16;
      case RATOM_CG: return ACE_15;
      default: return ACE_NONE;
     }
     break;
   case RESIDUE_LYN:
   case RESIDUE_LYS:
     switch (at) {
      case RATOM_N: return ACE_0;
      case RATOM_CA: return ACE_1;
      case RATOM_C: return ACE_2;
      case RATOM_O: return ACE_3;
      case RATOM_OXT: return ACE_3;
      case RATOM_CB: return ACE_5;
      case RATOM_CD: return ACE_7;
      case RATOM_CE: return ACE_6;
      case RATOM_CG: return ACE_15;
      case RATOM_NZ: return ACE_6;
     default: return ACE_NONE;
     }
     break;
   case RESIDUE_MET:
     switch (at) {
      case RATOM_N: return ACE_0;
      case RATOM_CA: return ACE_1;
      case RATOM_C: return ACE_2;
      case RATOM_O: return ACE_3;
      case RATOM_OXT: return ACE_3;
      case RATOM_CB: return ACE_5;
      case RATOM_CE: return ACE_16;
      case RATOM_CG: return ACE_15;
      case RATOM_SD: return ACE_15;
      default: return ACE_NONE;
     }
     break;
   case RESIDUE_PHE:
```

```
    switch (at) {
     case RATOM_N: return ACE_0;
     case RATOM_CA: return ACE_1;
     case RATOM_C: return ACE_2;
     case RATOM_O: return ACE_3;
     case RATOM_OXT: return ACE_3;
     case RATOM_CB: return ACE_5;
     case RATOM_CD1: return ACE_15;
     case RATOM_CD2: return ACE_15;
     case RATOM_CE1: return ACE_15;
     case RATOM_CE2: return ACE_15;
     case RATOM_CG: return ACE_15;
     case RATOM_CZ: return ACE_15;
     default: return ACE_NONE;
    }
    break;
  case RESIDUE_PRO:
    switch (at) {
     case RATOM_N: return ACE_0;
     case RATOM_CA: return ACE_1;
     case RATOM_C: return ACE_2;
     case RATOM_O: return ACE_3;
     case RATOM_OXT: return ACE_3;
     case RATOM_CB: return ACE_5;
     case RATOM_CD: return ACE_5;
     case RATOM_CG: return ACE_5;
     default: return ACE_NONE;
    }
    break;
  case RESIDUE_SER:
    switch (at) {
     case RATOM_N: return ACE_0;
     case RATOM_CA: return ACE_1;
     case RATOM_C: return ACE_2;
     case RATOM_O: return ACE_3;
     case RATOM_OXT: return ACE_3;
     case RATOM_CB: return ACE_12;
     case RATOM_OG: return ACE_12;
     default: return ACE_NONE;
    }
    break;
  case RESIDUE_THR:
    switch (at) {
     case RATOM_N: return ACE_0;
     case RATOM_CA: return ACE_1;
     case RATOM_C: return ACE_2;
     case RATOM_O: return ACE_3;
     case RATOM_OXT: return ACE_3;
     case RATOM_CB: return ACE_5;
     case RATOM_CG2: return ACE_15;
     case RATOM_OG1: return ACE_12;
     default: return ACE_NONE;
    }
    break;
  case RESIDUE_TRP:
    switch (at) {
     case RATOM_N: return ACE_0;
     case RATOM_CA: return ACE_1;
     case RATOM_C: return ACE_2;
     case RATOM_O: return ACE_3;
     case RATOM_OXT: return ACE_3;
     case RATOM_CB: return ACE_5;
     case RATOM_CD1: return ACE_15;
     case RATOM_CD2: return ACE_15;
     case RATOM_CE2: return ACE_15;
     case RATOM_CE3: return ACE_15;
     case RATOM_CG: return ACE_15;
     case RATOM_CH2: return ACE_15;
     case RATOM_CZ2: return ACE_15;
     case RATOM_CZ3: return ACE_15;
     case RATOM_NE1: return ACE_13;
     default: return ACE_NONE;
    }
    break;
  case RESIDUE_TYR:
    switch (at) {
     case RATOM_N: return ACE_0;
     case RATOM_CA: return ACE_1;
     case RATOM_C: return ACE_2;
     case RATOM_O: return ACE_3;
     case RATOM_OXT: return ACE_3;
     case RATOM_CB: return ACE_5;
     case RATOM_CD1: return ACE_15;
     case RATOM_CD2: return ACE_15;
     case RATOM_CE1: return ACE_14;
     case RATOM_CE2: return ACE_14;
     case RATOM_CG: return ACE_15;
     case RATOM_CZ: return ACE_14;
     case RATOM_OH: return ACE_12;
     default: return ACE_NONE;
```

```
      }
      break;
    case RESIDUE_VAL:
      switch (at) {
       case RATOM_N: return ACE_0;
       case RATOM_CA: return ACE_1;
       case RATOM_C: return ACE_2;
       case RATOM_O: return ACE_3;
       case RATOM_OXT: return ACE_3;
       case RATOM_CB: return ACE_5;
       case RATOM_CG1: return ACE_16;
       case RATOM_CG2: return ACE_16;
       default: return ACE_NONE;
      }
      break;
    default: return ACE_NONE;
   }
}


double ACEenergy(Residuetype rt1, ResidueAtomtype at1,
                 Residuetype rt2, ResidueAtomtype at2 ) {
   ACEtype t1;
   ACEtype t2;

   t1=getACEtype(rt1,at1);
   t2=getACEtype(rt2,at2);

   if (t1==ACE_NONE) return 0.0;
   if (t2==ACE_NONE) return 0.0;
   return ACEinteraction[t1][t2];
}

Complex getACEfield(Residuetype rt1, ResidueAtomtype at1) {
   ACEtype t1;
   Complex c;
   c.re=0;
   c.im=0;
   t1=getACEtype(rt1,at1);
   if (t1==ACE_NONE) return c;
   return ACEfield[t1];
}

Complex getACEdensity(Residuetype rt1, ResidueAtomtype at1) {
   ACEtype t1;
   Complex c;
   c.re=0;
   c.im=0;
   t1=getACEtype(rt1,at1);
   if (t1==ACE_NONE) return c;
   return ACEdensity[t1];
}
```

rotamer.hh

```
// *************************************************
// File: rotamer.hh
// purpose: rotamer library manipulation and query
//
// author: Hans-Henrik Staerfeldt
// notes:

#include <stdio.h>

#define ROTAMER_MAXTYPE 32
#define ROTAMER_MAXID 4
#define ROTAMER_VERSION "ROTAMERLIBRARY 1.0"

// *************************************************
// struct: rotamerrecord
// purpose: record in the rotamer library
// notes:

typedef struct rotamerrecord_ {
    char ident[ROTAMER_MAXID];
    float phi;                                   /* backbone conformation */
    float psi;                                   /* backbone conformation */
    int N;
    int r[4];
    float p;
    float chi[4];                                /* rotation angles */
} rotamerrecord;

// *************************************************
// struct: rotamerheader
// purpose: header of the rotamer library
// notes:

typedef struct rotamerheader_ {
    char version[64];
    long maxtype;
    long seekpos[ROTAMER_MAXTYPE];
    long count[ROTAMER_MAXTYPE];
    char ident[ROTAMER_MAXTYPE][ROTAMER_MAXID];
} rotamerheader;

// *************************************************
// class: rotamerlibrary
// purpose: rotamer library class
// notes:

class rotamerlibrary {
  public:
    ~rotamerlibrary(void);
    rotamerlibrary(char *filename);
    rotamerlibrary(void);

    int open(char *filename);
    int close(void);
    int report(void);

    int set_type(char *ident);
    int set_index(int index);
    int get_record(rotamerrecord *rec);
    long get_count(char *ident);
    long get_count(void);

    int convert(char *infilename, char *outfilename);

  private:
    int currenttypeidx;
    rotamerheader header;
    FILE *f;
};
```

rotamer.cc

```
// *************************************************
// File: rotamer.cc
// purpose: rotamer library implementation
//
// author: Hans-Henrik Staerfeldt
// notes:

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <stream.h>
#include "rotamer.hh"

int rotamerlibrary::report(void) {
    int i;
    cout << header.version << endl;
    cout << "Maxtype:  "<<header.maxtype<<" Record size:";
    cout << sizeof(rotamerheader)<<endl;
    for (i=0;i<header.maxtype;i++) {
        cout << "Type:" << header.ident[i] << " Seekpos:";
        cout << header.seekpos[i];
        cout << " Count:" << header.count[i] << endl;
    }
    return 0;
}

int rotamerlibrary::convert(char *infilename, char *outfilename){
    FILE *inf;
    FILE *outf;
    size_t r;
    long linenumber=0;
    rotamerrecord rec;
    char currenttype[4]="\\0\\0\\0";

    inf=fopen(infilename,"r");
    if (!inf)
      throw "could not open text rotamer library";

    outf=fopen(outfilename,"w");
    if (!outf)
      throw "could not open binary rotamer library";

    strcpy(header.version,ROTAMER_VERSION);
    header.maxtype=0;

    /* make room for header */
    fseek(outf,sizeof(rotamerheader),SEEK_SET);

    /* first scan ascii version for numbers while converting */
    while (!feof(inf)){
        r=fscanf(inf,"%3s %f %f %d %d %d %d %d %f %f %f %f %f\n",
                 (char *)rec.ident,
                 &rec.phi,
                 &rec.psi,
                 &rec.N,
                 &rec.r[0],
                 &rec.r[1],
                 &rec.r[2],
                 &rec.r[3],
                 &rec.p,
                 &rec.chi[0],
                 &rec.chi[1],
                 &rec.chi[2],
                 &rec.chi[3]);
        linenumber++;
        if (r!=13) throw "could not read line of rotamer library";

        /* book keeping */
        if (!*currenttype){
            strcpy((char *)currenttype,(char *)rec.ident);
            strcpy((char *)header.ident[header.maxtype],(char *)rec.ident);
            header.seekpos[header.maxtype]=ftell(outf);
        }

        header.count[header.maxtype]++;

        if (strcmp((char *)rec.ident,(char *)currenttype)) {
            /* new type */
            header.maxtype++;
            strcpy((char *)header.ident[header.maxtype],(char *)rec.ident);
            strcpy((char *)currenttype,(char *)rec.ident);
            header.seekpos[header.maxtype]=ftell(outf);
        }

        r=fwrite(&rec,1,sizeof(rotamerrecord),outf);
        if (r!=sizeof(rotamerrecord))
          throw "could not write line of rotamer library";
    }

    /* end of read */
    header.count[header.maxtype]++;
    header.maxtype++;

    /* .. then write header */
    fseek(outf,0,SEEK_SET);
    r=fwrite(&header,1,sizeof(rotamerheader),outf);
    if (r!=sizeof(rotamerheader))
      throw "could not write header of rotamer library";

    fclose(outf);
    fclose(inf);
    return 0;
}
```

```
int rotamerlibrary::open(char *filename) {
   size_t r;

   if (f)
     throw "rotamer library already open";

   /* open file and read library header */
   memset(&header,0,sizeof(rotamerheader));

   f=fopen(filename,"r");
   if (!f) throw "could not open rotamer library";

   r=fread(&header,1,sizeof(rotamerheader),f);
   if (r≠sizeof(rotamerheader))
     throw "could not read header of rotamer library";

   if (strcmp(header.version,ROTAMER_VERSION))
     throw "wrong version of rotamer library";

   currenttypeidx=0;
   return 0;
}

int rotamerlibrary::close(void) {
   if (f) return fclose(f);
   return 0;
}

rotamerlibrary::rotamerlibrary(char *filename) {
   f=NULL;
   open(filename);
}

rotamerlibrary::rotamerlibrary(void) {
   /* empty library */
   memset(&header,0,sizeof(rotamerheader));
   f=NULL;
   currenttypeidx=0;
}

rotamerlibrary::~rotamerlibrary(void) {
   close();
}

int rotamerlibrary::set_type(char *ident) {
   int r;
   int i;
   // This is an _ugly_ hack for equivalence residues
   char *eq[]={"CYX","CYH","CYS",
               "ASH","ASP","GLH",
               "GLU","HID","HIE",
               "HIP","HIS","LYN",
               "LYS",NULL};

   for (r=0;r<header.maxtype;r++) {
      if (!strcmp(ident,(char *)header.ident[r])) {
         fseek(f,header.seekpos[r],SEEK_SET);
         currenttypeidx=r;
         return 0;
      }
   }

   for (i=0;eq[i];i++){
      if (!strcmp(ident,eq[i])) {
         // we have a eq match
         for (i=i+1;eq[i];i++){ // search for the eq (first match)
            for (r=0;r<header.maxtype;r++) {
               if (!strcmp(eq[i],(char *)header.ident[r])) {
                  fseek(f,header.seekpos[r],SEEK_SET);
                  currenttypeidx=r;
                  return 0;
               }
            }
         }
         break; // search only first match (duh)
      }
   }


   return -1;
}

int rotamerlibrary::set_index(int index) {
   /* end of type records? */
   if (index≥header.count[currenttypeidx])
     throw "end of records";
   return fseek(f,header.seekpos[currenttypeidx]+
              sizeof(rotamerrecord)*index,SEEK_SET);
}

int rotamerlibrary::get_record(rotamerrecord *rec) {
   size_t r;

   if (!f) throw "rotamer library not open";

   if (feof(f)) throw "end of records";

   /* check index for validity */
   r=(ftell(f)-header.seekpos[currenttypeidx])/sizeof(rotamerrecord);

   /* end of type records? */
   if (r≥header.count[currenttypeidx])
     throw "end of records";

   r=fread(rec,1,sizeof(rotamerrecord),f);
   if (r<sizeof(rotamerrecord))
     throw "could not read record";

   return 0;
}

long rotamerlibrary::get_count(char *ident) {
   int r;
   if (!f) throw "rotamer library not open" ;
   for (r=0;r<header.maxtype;r++) {
      if (!strcmp(ident,(char *)header.ident[r])) {
         return header.count[r];
      }
   }
   throw "illegal residue type";
}

long rotamerlibrary::get_count(void) {
   if (!f) throw "rotamer library not open";
   return header.count[currenttypeidx];
}
```

grid.hh

```
// ************************************************
// File: grid.hh
// purpose: complex grids for holding sampled
// density or field functions
// author: Hans-Henrik Staerfeldt
// notes:

#ifndef __RENDERER_H__
#define __RENDERER_H__

#include <vector>
using namespace std;
#include <fftw.h>
#include <rfftw.h>
#include "vector3d.hh"
#include "complex.hh"

#define GRIDSIZE 128

class CGridPoint {
   Vector3D pos;
   Complex v;
};

// ************************************************
// class: Grid
// purpose: virtual class common to all grid types
// notes: currently only complex grid exist as subtypes

class Grid {
   friend class CGrid;
   friend istream &operator>>(istream &s,CGrid &a);
   friend ostream &operator<<(ostream &s,const CGrid &a);
 private:
   int size;
   int sizesquared;
   int sizecubed;
   int sizehalf;
   double scale; // scale of map Å/pixel

 public:

   void init(const double iscale,const int isize);
   Grid();
   Grid(double iscale,int isize); // construct with given scale and size

   inline int getidx(const int x,const int y,const int z) const
   {
      int nx,ny,nz;
      nx=x%size; ny=y%size; nz=z%size;
```

```
    return nx*sizesquared+ny*size+nz;
  };
  inline int getsize() const { return size; }
  inline int getsizesquared() const { return sizesquared; }
  inline int getsizecubed() const { return sizecubed; }
  inline int getsizehalf() const { return sizehalf; }
  inline double getscale() const { return scale; }
  inline void setscale(double iscale) { scale=iscale; }

  inline Vector3D getvector(const int x,const int y,const int z) const
  {Vector3D r(x*scale,y*scale,z*scale);return r;};
  inline Vector3D getvector(const int index) const
  {int x,y,z; getposition(index,x,y,z); return getvector(x,y,z);}

  inline void getposition(const int index, int &x, int &y, int &z) const
  {
    int idx;
    idx=index; z=(idx%size); idx-=z;
    idx/=size; y=(idx%size); idx-=y;
    idx/=size; x=(idx%size); idx-=x;
  }

  void getclippedbox(const Vector3D center,
                     const double pixelwidth,
                     int min[3], int max[3]);

};

// ************************************************
// class: CGrid
// purpose: grid with complex value cells
// notes:

class CGrid :public Grid {

  Complex *map; // pixels
  fftwnd_plan forward_plan;
  fftwnd_plan backward_plan;

 public:
  void init(const double iscale,const int isize);
  CGrid(); // construct with no scale and size
  CGrid(double scale,int size); // construct with given scale and size
  ~CGrid();

  inline Complex &operator()(const int x,const int y,const int z)
  { return map[getidx(x,y,z)]; };
  inline Complex &operator()(const int index)
  { return map[index]; }
  inline Complex operator()(const int x,const int y,const int z) const
  { int idx=getidx(x,y,z); return map[idx]; };
  inline Complex operator()(const int index) const
  { return map[index]; }

  inline void setmapre(const int index,const double re)
  { map[index].re=re;};
  inline void setmapim(const int index,const double im)
  { map[index].im=im;};
  inline void addmapre(const int index,const double re)
  { map[index].re+=re;};
  inline void addmapim(const int index,const double im)
  { map[index].im+=im;};
  inline void setmap(const int index,const Complex c)
  { map[index]=c;};

  void clear(const double r, const double i); // clear the map
  void clear();
  void compl();
  void recenter();
  void translate(const int dx, const int dy, const int dz);
  void scaledown(); // scales down with size of grid
  void transformfield(const CGrid &src,
                      const Vector3D &offset,
                      const Vector3D &srcxv,
                      const Vector3D &srcyv,
                      const Vector3D &srczv);
  void transformdelta(const CGrid &src,
                      const Vector3D &offset,
                      const Vector3D &srcxv,
                      const Vector3D &srcyv,
                      const Vector3D &srczv);

  void getstat(Complex &min, Complex &max,
               Complex &avg, Complex &dev, Complex &dev0,
               double &minamp, double &maxamp,
               double &avgamp, double &devamp, double &devamp0);

  // write slice in gnuplot data format (real, imag)
  void writesliceregp(int x, ostream &s);
  void writesliceimgp(int x, ostream &s);

  void save(char *filename) ;
  void load(char *filename) ;

  // merge two maps into this one, using maximum
  void mergemax(const CGrid &a,const CGrid &b);
  // merge two maps into this one, using minimum
  void mergemin(const CGrid &a,const CGrid &b);
  // multiply two maps into this one
  void multiply(const CGrid &a,const CGrid &b);
  // add two maps into this one
  void add(const CGrid &a,const CGrid &b);
  // take fft of this map
  void fft();
  // take ifft of this map
  void ifft();
  void corr(CGrid &a, CGrid &b);

  CGrid &operator=(const CGrid &a);

  friend istream &operator>>(istream &s,CGrid &a);
  friend ostream &operator<<(ostream &s,const CGrid &a);
};

istream &operator>>(istream &s,CGrid &a);
ostream &operator<<(ostream &s,const CGrid &a);


#endif
```

grid.cc

```
// ************************************************
// File: grid.cc
// purpose: complex grids for holding sampled
// density or field functions
// author: Hans-Henrik Staerfeldt
// notes:

#include "vector3d.hh"
#include "field.hh"
#include "grid.hh"
#include <limits.h>
#include <math.h>

// #define ASCIIMODE

// #define DEBUG

// ************************************************************
// General Grid
// ************************************************************

void Grid::init(const double iscale,const int isize) {
  size=isize;
  sizehalf=isize/2;
  scale=iscale;
  sizesquared=isize*isize;
  sizecubed=sizesquared*isize;
}

Grid::Grid(double iscale,int isize) {
  init(iscale,isize);
}

Grid::Grid() {
  init(1.0,0);
}

void Grid::getclippedbox(const Vector3D center,
                         const double pixelwidth,
                         int min[3], int max[3]) {
  Vector3D scaledcenter;

  scaledcenter=center/scale;

  // calculate ranges in pixel space

  min[0]=scaledcenter[0]-pixelwidth;
  if (min[0]<0) min[0]=0;
  min[1]=scaledcenter[1]-pixelwidth;
  if (min[1]<0) min[1]=0;
  min[2]=scaledcenter[2]-pixelwidth;
  if (min[2]<0) min[2]=0;

  max[0]=scaledcenter[0]+pixelwidth+1;
  if (max[0]>size) max[0]=size;
  max[1]=scaledcenter[1]+pixelwidth+1;
  if (max[1]>size) max[1]=size;
  max[2]=scaledcenter[2]+pixelwidth+1;
```

```
    if (max[2]>size) max[2]=size;

}

// ***********************************************************
// Grid with complex values
// ***********************************************************

CGrid::CGrid() {
    Grid::init(1.0,0);
    map = NULL;
    forward_plan = NULL;
    backward_plan = NULL;
}

CGrid::CGrid(const double iscale,const int isize) {
    init(iscale,isize);
}

void CGrid::init(const double iscale,const int isize) {
    Grid::init(iscale,isize);
    if(size) {
        map=new Complex[sizecubed];
        if (map==NULL) throw "Unable to allocate map";
        forward_plan = fftw3d_create_plan(isize, isize, isize, FFTW_FORWARD,
                                          FFTW_ESTIMATE |
FFTW_IN_PLACE);
        backward_plan = fftw3d_create_plan(isize, isize, isize, FFTW_BACKWARD,
                                           FFTW_ESTIMATE |
FFTW_IN_PLACE);
        clear();
    }else{
        map = NULL;
        forward_plan = NULL;
        backward_plan = NULL;
    }
}

CGrid::~CGrid(){
    if (map) delete map;
    if (forward_plan) fftwnd_destroy_plan(forward_plan);
    if (backward_plan) fftwnd_destroy_plan(backward_plan);
}

void CGrid::clear(){
    memset(map,0,sizeof(Complex)*sizecubed);
}

void CGrid::clear(const double r, const double i){
    // clear the map
    int idx;
    Complex c;
    c.re=r;
    c.im=i;
    for (idx=0;idx<sizecubed;idx++)
      map[idx]=c;
}

void CGrid::recenter() {
    int sh=sizehalf;
    translate(sh,sh,sh);
}

CGrid &CGrid::operator=(const CGrid &a) {
    if (a.size≠size || a.scale≠scale) {
        if (map) free(map);
        if (forward_plan) fftwnd_destroy_plan(forward_plan);
        if (backward_plan) fftwnd_destroy_plan(backward_plan);
        init(a.scale,a.size);
    }
    memcpy(map,a.map,sizecubed*sizeof(Complex));
    return *this;
}

void CGrid::save(char *filename) {
    ofstream ofs;
    ofs.open(filename);
    if (ofs.bad()) throw "Could not write grid";
    ofs≪*this;
    ofs.close();
}

void CGrid::load(char *filename) {
    ifstream ifs;
    ifs.open(filename);
    if (ifs.bad()) throw "Could not read grid";
    ifs≫*this;
    ifs.close();
}

// recenter grid for pretty print...
void CGrid::translate(const int dx, const int dy, const int dz) {
    int i,j;
    int x,y,z;
    int nx,ny,nz;
    Complex c;

    nx=dx%size;
    ny=dy%size;
    nz=dz%size;

    for (i=0,x=0;x<size;x++){
        for (y=0;y<size;y++){
            for (z=0;z<size;z++,i++){
                j=getidx((x+nx)%size,(y+ny)%size,(z+nz)%size);
                if (i>j) continue;
                c=map[i];
                map[i]=map[j];
                map[j]=c;
            }
        }
    }

}

void CGrid::corr(CGrid &a, CGrid &b) {

    if (&a==&b || &a==this || &b==this)
      throw "Arguments to correlation must be distict";

    // a.recenter();
    a.ifft();
    // b.recenter();
    b.fft();

    if (scale≠a.scale || size≠a.size)
      init(a.scale,a.size);

    multiply(a, b);
    ifft();

    int i,j;
    int x,y,z;
    double scaledown=1.0/double(sizecubed);
    Complex c;

    // translation reversal, rescaling in one go.
    for (i=0,x=0;x<size;x++){
        for (y=0;y<size;y++){
            for (z=0;z<size;z++,i++){
                j=a.getidx(size-(x+sizehalf)%size,\
                           size-(y+sizehalf)%size,
                           size-(z+sizehalf)%size);
                if (i>j) continue;
                c=map[i];
                map[i]=map[j]*scaledown;
                map[j]=c*scaledown;
            }
        }
    }
}



static double sqr(double d) { return d*d; };

void CGrid::getstat(Complex &min, Complex &max, Complex &avg,
                    Complex &dev, Complex &dev0,
                    double &minamp, double &maxamp,
                    double &avgamp, double &devamp, double &devamp0) {
    int i;
    double amp;
    Complex c;

    max.re=-DBL_MAX;
    min.re=DBL_MAX;
    max.im=-DBL_MAX;
    min.im=DBL_MAX;
    maxamp=-DBL_MAX;
    minamp=DBL_MAX;
    avg.re=0.0;
    avg.im=0.0;
    avgamp=0.0;

    dev.re=1.0;
    dev.im=1.0;
    dev0.re=1.0;
    dev0.im=1.0;
    devamp=1.0;
    devamp0=1.0;
```

```
    for (i=0;i<sizecubed;i++) {
        c=map[i];
        amp=amplitude(map[i]);
        avgamp+=amp;
        avg.re+=c.re;
        avg.im+=c.im;
        if (amp>maxamp) maxamp=amp;
        if (amp<minamp) minamp=amp;
        if (c.re>max.re)max.re=c.re;
        if (c.re<min.re)min.re=c.re;
        if (c.im>max.im)max.im=c.im;
        if (c.im<min.im)min.im=c.im;
    }

    avg.re/=double(sizecubed);
    avg.im/=double(sizecubed);
    avgamp/=double(sizecubed);

    if (sizecubed<2) return;

    dev.re=0.0;
    dev.im=0.0;
    dev0.re=0.0;
    dev0.im=0.0;
    devamp=0.0;
    devamp0=0.0;

    for (i=0;i<sizecubed;i++) {
        c=map[i];
        amp=amplitude(map[i]);
        dev.re += sqr(avg.re - c.re);
        dev.im += sqr(avg.im - c.im);
        dev0.re += sqr(c.re);
        dev0.im += sqr(c.im);
        devamp += sqr(avgamp-amp);
        devamp0 += sqr(amp);
    }

    dev0.re/=double(sizecubed-1);
    dev0.im/=double(sizecubed-1);
    dev.re/=double(sizecubed-1);
    dev.im/=double(sizecubed-1);
    devamp/=double(sizecubed-1);
    devamp0/=double(sizecubed-1);
}

// complement
void CGrid::compl(){
    int i;
    for (i=0;i<sizecubed;i++)
       map[i].im=-map[i].im;
}


// ************************************************
// method: CGrid::transformfield
// purpose: Render a rotated/translated source
// grid with a field onto this grid.
// notes:
//
// The grids are assumed not to have edge effects, fields in this
// grid outside the rotated grid (src) have value 0.0
// Grids are assumed to be centered at (sizehalf,sizehalf,sizehalf)
// extending sizehalf in each direction
// Merging is done by averaging on 'src' grid cells with a grid cell
// radius.
// Handles differences in scale and size
void CGrid::transformfield(const CGrid &src,
                           const Vector3D &offset,
                           const Vector3D &srcxv,
                           const Vector3D &srcyv,
                           const Vector3D &srczv) {
    // src: source grid
    // srcx,srcy,srcz: source grid normal vectors in destination space.

    Vector3D dstcx; // grid corner x in destination grid
    Vector3D dstcy; // grid corner y in destination grid
    Vector3D dstcz; // grid corner z in destination grid

    Vector3D dstxv;
    Vector3D dstyv;
    Vector3D dstzv;

    int srcx,srcy,srcz;
    int dstx,dsty,dstz;
    Vector3D unitx(1.0,0.0,0.0);
    Vector3D unity(0.0,1.0,0.0);
    Vector3D unitz(0.0,0.0,1.0);
    Vector3D zero(0.0,0.0,0.0);
    Vector3D dstgridoffset;
    Matrix4D mat;
    int dstidx;
    int srcidx;
    double rescale;
    int srccellwidth;
    int srccellwb;
    int srccellwe;
    Vector3D srcp; // source point

    // The rescale tells how much longer/shorter a step in the dst grid
    // is compared to the src grid
    rescale=scale/src.scale;
    srccellwidth=floor(rescale/2.0);
    srccellwb=floor(rescale/3.0);
    srccellwe=ceil(rescale/3.0);

    // calculate destination unit vectors,
    // at zero, look along z vector, y is up
    mat.viewat(zero,srczv,srcyv);
    dstxv=(unitx*mat)*rescale;
    dstyv=(unity*mat)*rescale;
    dstzv=(unitz*mat)*rescale;

    // offset is in the scale and space of the destination
    // and need to be transformed into the gridspace of the source
    dstgridoffset=((-offset/scale)*mat)*rescale;

    // calculate dst corner vectors
    dstcx=-dstxv*sizehalf;
    dstcy=-dstyv*sizehalf;
    dstcz=-dstzv*sizehalf;

#ifdef DEBUG
    cerr<<"true offset of source;"<<offset<<endl;
    cerr<<"destination grid offset of source;"<<offset/scale<<endl;
    cerr<<"source grid offset of destination"<<dstgridoffset<<endl;
    cerr<<"source system;"<<endl<<srcxv<<endl<<srcyv<<endl<<srczv<<endl;
    cerr<<"rescale;"<<rescale<<endl;
    cerr<<"srccellwidth;"<<-srccellwb<<".."<<srccellwe<<endl;
    cerr<<"destination system;"<<endl<<dstxv<<endl<<dstyv<<endl<<dstzv<<endl;
    cerr<<"destination corner;"<<endl<<dstcx<<endl<<dstcy<<endl<<dstcz<<endl;
    cerr<<"destination corner;"<<endl<<dstcx+dstcy+dstcz<<endl;
#endif

    // loop over destination
    for (dstidx=0,dstx=0;dstx<size;dstx++){
        for (dsty=0;dsty<size;dsty++){
            srcp=dstcx+dstcy+dstcz+dstgridoffset;
            for (dstz=0;dstz<size;dstz++,dstidx++){
                srcp+=dstzv; // step along zaxis
                if (srcp[0]<-src.sizehalf) continue;
                if (srcp[0]>=src.sizehalf) continue;
                if (srcp[1]<-src.sizehalf) continue;
                if (srcp[1]>=src.sizehalf) continue;
                if (srcp[2]<-src.sizehalf) continue;
                if (srcp[2]>=src.sizehalf) continue;

                srcx=int(srcp[0])+src.sizehalf;
                srcy=int(srcp[1])+src.sizehalf;
                srcz=int(srcp[2])+src.sizehalf;

                if (srccellwidth>=1) {
                    int count=0;
                    int i,j,k;
                    Complex sum;
                    sum.re=0.0;
                    sum.im=0.0;
                    // anitaliasing if compressed
                    for (i=srcx-srccellwb;i<srcx+srccellwe;i++){
                        if (i<0) continue;
                        if (i>src.size) continue;
                        for (j=srcy-srccellwb;j<srcy+srccellwe;j++){
                            if (j<0) continue;
                            if (j>src.size) continue;
                            for (k=srcz-srccellwb;k<srcz+srccellwe;k++){
                                if (k<0) continue;
                                if (k>src.size) continue;

                                srcidx=src.getidx(i,j,k);
                                sum=sum+src.map[srcidx];
                                count++;
                            }
                        }
                    }

                    if (count==0) count=1;

                    map[dstidx].re=sum.re/double(count);
                    map[dstidx].im=sum.im/double(count);
```

```
            }else{
                // just one value
                srcidx=src.getidx(srcx,srcy,srcz);
                map[dstidx]=src.map[srcidx]; // no antialiasing
            }

        }
        dstcy+=dstyv; // step along yaxis
      }
      dstcy-=dstyv*size;
      dstcx+=dstxv; // step along xaxis
   }
}


// ***************************************************
// method: CGrid::transformdelta
// purpose: Render a rotated/translated source grid
// with _delta_ function onto this grif
// notes:
//
// As delta functions occur in single points in the source
// and may not be smoothed.
// In fact they must occur as a sum in the resulting rotated grid.
// Therefore this rotation loops over the source grid points instead
// of the destination grid points. Important of the delta grid is
// scaled up.
// The grids are assumed not to have edge effects, fields in this
// grid outside the rotated grid (src) have value 0.0
// Grids are assumed to be centered at (sizehalf,sizehalf,sizehalf)
// extending sizehalf in each direction
// Merging is done by averaging on 'src' grid cells with a grid cell
// radius.
// Handles difference in scale and size
void CGrid::transformdelta(const CGrid &src,
                           const Vector3D &offset,
                           const Vector3D &srcxv,
                           const Vector3D &srcyv,
                           const Vector3D &srczv) {
   // src: source grid
   // srcx,srcy,srcz: source grid normal vectors in destination space.

   Vector3D srccx; // grid corner x in destination grid
   Vector3D srccy; // grid corner y in destination grid
   Vector3D srccz; // grid corner z in destination grid
   int srcx,srcy,srcz;
   int dstx,dsty,dstz;
   Vector3D srcxvs;
   Vector3D srcyvs;
   Vector3D srczvs;
   int dstidx;
   int srcidx;
   double rescale;
   Vector3D srcgridoffset;
   Vector3D dstp; // destination point

   rescale=scale/src.scale;
   srcgridoffset=offset/scale;

   // calculate corner of source
   srcxvs=srcxv/rescale;
   srcyvs=srcyv/rescale;
   srczvs=srczv/rescale;

   srccx=-srcxvs*src.sizehalf;
   srccy=-srcyvs*src.sizehalf;
   srccz=-srczvs*src.sizehalf;

#ifdef DEBUG
   cerr<<"rescale;"<<rescale<<endl;
   cerr<<"destination offset of source;"<<offset<<endl;
   cerr<<"destination grid offset of source;"<<srcgridoffset<<endl;
   cerr<<"Source system;"<<endl<<srcxv<<endl<<srcyv<<endl<<srczv<<endl;
   cerr<<"Source corner;"<<endl<<srccx<<endl<<srccy<<endl<<srccz<<endl;
   cerr<<"Source corner;"<<endl<<srccx+srccy+srccz<<endl;
#endif

   // loop over source
   for (srcidx=0,srcx=0;srcx<src.size;srcx++){
      for (srcy=0;srcy<src.size;srcy++){
         dstp=srccx+srccy+srccz+srcgridoffset;
         for (srcz=0;srcz<src.size;srcz++,srcidx++){
            dstp+=srczvs; // step along zaxis
            if (dstp[0]<-sizehalf) continue;
            if (dstp[0]>=sizehalf) continue;
            if (dstp[1]<-sizehalf) continue;
            if (dstp[1]>=sizehalf) continue;
            if (dstp[2]<-sizehalf) continue;
            if (dstp[2]>=sizehalf) continue;
            dstx=int(dstp[0])+sizehalf;
            dsty=int(dstp[1])+sizehalf;
            dstz=int(dstp[2])+sizehalf;
            dstidx=getidx(dstx,dsty,dstz);
            // no antialiasing but sum
            map[dstidx]=map[dstidx]+src.map[srcidx];
         }
         srccy+=srcyvs; // step along yaxis
      }
      srccy-=srcyvs*src.size;
      srccx+=srcxvs; // step along xaxis
   }
}

// merge two maps into this one, using maximum
void CGrid::mergemax(const CGrid &a,const CGrid &b){
   int i;
   for (i=0;i<sizecubed;i++)
     map[i]=maxamp(a.map[i],b.map[i]);
}

// merge two maps into this one, using minimum
void CGrid::mergemin(const CGrid &a,const CGrid &b){
   int i;
   for (i=0;i<sizecubed;i++)
     map[i]=minamp(a.map[i],b.map[i]);
}

// multiply two maps into this one
void CGrid::multiply(const CGrid &a,const CGrid &b){
   int i;
   for (i=0;i<sizecubed;i++)
     map[i]=a.map[i]*b.map[i];
}

// scale down with size of grid
void CGrid::scaledown(){
   int i;
   for (i=0;i<sizecubed;i++){
      map[i].re=map[i].re/double(sizecubed);
      map[i].im=map[i].im/double(sizecubed);
   }
}

// add two maps into this one
void CGrid::add(const CGrid &a,const CGrid &b){
   int i;
   for (i=0;i<sizecubed;i++)
     map[i]=a.map[i]+b.map[i];
}

// take fft of this map
void CGrid::fft(void){
   // make fast fourier transform
   fftwnd_one(forward_plan, map, NULL);
}

// take ifft of this map
void CGrid::ifft(void){
   // make inverse fast fourier transform
   fftwnd_one(backward_plan, map, NULL);
}

// write slice in gnuplot data format (real)
void CGrid::writesliceregp(int x, ostream &s) {
   int i,y,z;

   i=x*sizesquared;
   for (y=0;y<size;y++) {
      for (z=0;z<size;z++,i++) {
         s << map[i].re << "\n";
      }
      s << "\n";
   }
}

// write slice in gnuplot data format (imag)
void CGrid::writesliceimgp(int x, ostream &s) {
   int i,y,z;

   i=x*sizesquared;
   for (y=0;y<size;y++) {
      for (z=0;z<size;z++,i++) {
         s << map[i].im << "\n";
      }
      s << "\n";
   }
}

istream &operator>>(istream &s, CGrid &a) {
   int l;
```

```
    char mode;
    int i;

    s >> a.scale;
    s >> a.size;
    s >> mode;
    s.get(); // eat off the newline

    a.sizecubed=a.size*a.size*a.size;
    a.sizesquared=a.size*a.size;
    a.forward_plan = fftw3d_create_plan(a.size,a.size, a.size, FFTW_FORWARD,
                                                   FFTW_ESTIMATE | FFTW_IN_PLACE);
    a.backward_plan = fftw3d_create_plan(a.size, a.size, a.size, FFTW_BACKWARD,
                                                    FFTW_ESTIMATE |
FFTW_IN_PLACE);
    l=a.sizecubed;

    if (a.map) delete a.map;
    a.map=new Complex[l];

    if (mode=='A') {
        for (i=0;i<l;i++) {
            s >> a.map[i].re;
            s >> a.map[i].im;
        }
    }else if (mode=='B'){
        s.read((char *)a.map,l*sizeof(Complex));
    }
    return s;
}

//todo: make it binary
ostream &operator<<(ostream &s,const CGrid &a) {
    int l;
    l=a.sizecubed;

    s << a.scale << " ";
    s << a.size << " ";

#ifdef ASCIIMODE
    s << "A" << endl; // ascii mode
    int i;
    int l1=0;
    int l2=0;
    int sizesqared=a.size*a.size;
    for (i=0;i<l;i++,l1++,l2++){
        if (l1==a.size) { s << "\n"; l1=0;};
        if (l2==sizesqared) { s << "\n"; l2=0;};
        s << a.map[i].re << a.map[i].im << endl;
    }
#else
    s << "B" << endl; // binary mode
    s.write((char *)a.map,l*sizeof(Complex));
#endif

    return s;
}
```

gridset.hh

```
// ************************************************
// File: gridset.hh
// purpose: set of grids that hold the sampled
// field and density functions
// author: Hans-Henrik Staerfeldt
// notes:

#ifndef __GRIDSET_H__
#define __GRIDSET_H__

#include <vector>
using namespace std;

#include <fftw.h>
#include "complex.hh"
#include "grid.hh"

//#define DO_LJfield
#define DO_Desol
#define DO_Elec
#define DO_Shape

extern double LJfield_maxradius;// 6-12 potential field
extern double Desol_maxradius; // desolvation effects
extern double Elec_maxradius; // electrical field
extern double Shape_maxradius;
extern double Shape_rho;
extern double Shape_surface;

// ************************************************
// class: virtual class gridset
// purpose: common class for several different gridsets
// notes:

class GridSet {

 public:

    GridSet(double scale,int size);
    GridSet();
    ~GridSet(){};
    void init(double scale,int size);

    double scale;
    int size;
    int sizesquared;
    int sizecubed;
    int sizehalf;

    inline Vector3D getvector(const int x,const int y,const int z) const
    {Vector3D r(x*scale,y*scale,z*scale);return r;};

    inline void getposition(const int index, int &x, int &y, int &z) const
    {
        int idx;

        idx=index;
        z=(idx%size);
        idx-=z;

        idx/=size;
        y=(idx%size);
        idx-=y;

        idx/=size;
        x=(idx%size);
        idx-=x;
    }

    inline Vector3D getvector(const int index) const
    {
        int x,y,z;
        getposition(index,x,y,z);
        Vector3D r(x*scale,y*scale,z*scale);return r;
    }

    inline int getidx(const int x,const int y,const int z) const
    {
        int nx,ny,nz;
        nx=x%size;
        ny=y%size;
        nz=z%size;
        return nx*size*size+ny*size+nz;
    };

    void getclippedbox(const Vector3D center,
                       const double pixelwidth,
                       int min[3], int max[3]);
};

// ************************************************
// class: ReceptorGridSet
// purpose: gridset for the receptor firld and density functions
// notes:

class ReceptorGridSet : public GridSet{

 public:

#ifdef DO_LJfield
    CGrid LJfield;// 6-12 potential field
#endif

#ifdef DO_Desol
    CGrid DesolR; // desolvation effects receptor
#endif

#ifdef DO_Elec
    CGrid Elecfield; // electrical field
#endif

#ifdef DO_Shape
    CGrid ShapeR; // receptor part of shape
#endif

    CGrid Density; // density (delta function)
```

```
    ReceptorGridSet(double scale,int size);
    ReceptorGridSet(){};
    ~ReceptorGridSet(){};

    void clear( void );
    void rotate(const ReceptorGridSet &src,
                const Vector3D &srczv,
                const Vector3D &srcyv);
    void save(char *filename);
    void load(char *filename);
    void init(double scale,int size);
    inline int getidx(const int x,const int y,const int z) const
    {return GridSet::getidx(x,y,z);};
    inline void getclippedbox(const Vector3D center,
                              const double pixelwidth,
                              int min[3], int max[3])
    {GridSet::getclippedbox(center,pixelwidth,min,max);};
    inline Vector3D getvector(const int index) const
    { return GridSet::getvector(index);};
    inline Vector3D getvector(const int x,const int y,const int z) const
    { return GridSet::getvector(x,y,z);};
    inline void getposition(const int index, int &x, int &y, int &z) const
    {GridSet::getposition(index,x,y,z);};
    void ifft();
    void fft();
};

// *************************************************
// class: LigandGridSet
// purpose: gridset for the receptor firld and density functions
// notes: this is the gridset that gets rotated during search

class LigandGridSet : public GridSet{

 public:

#ifdef DO_Desol
    CGrid DesolL; // desolvation effects ligand, most,least
#endif

#ifdef DO_Elec
    CGrid Elecd; // electrical density
#endif

#ifdef DO_Shape
    CGrid ShapeL; // ligand part of shape
#endif

    CGrid Density;// density (delta function) most,least

    LigandGridSet(double scale,int size);
    LigandGridSet(){};
    ~LigandGridSet(){};

    void clear( void );
    void rotate(const LigandGridSet &src,
                const Vector3D &srczv,
                const Vector3D &srcyv);
    void save(char *filename);
    void load(char *filename);
    void init(double scale,int size);
    void merge(const LigandGridSet &src);

    inline int getidx(const int x,const int y,const int z) const
    {return GridSet::getidx(x,y,z);};
    inline void getclippedbox(const Vector3D center,
                              const double pixelwidth,
                              int min[3], int max[3])
    {GridSet::getclippedbox(center,pixelwidth,min,max);};
    inline Vector3D getvector(const int index) const
    { return GridSet::getvector(index);};
    inline Vector3D getvector(const int x,const int y,const int z) const
    { return GridSet::getvector(x,y,z);};
    inline void getposition(const int index, int &x, int &y, int &z) const
    {GridSet::getposition(index,x,y,z);};
    void ifft();
    void fft();
};


#endif
```

gridset.cc

```
// *************************************************
// File: gridset.cc
// purpose: set of grids that hold the sampled
// field and density functions
// author: Hans-Henrik Staerfeldt
// notes:

#include "gridset.hh"
#include "field.hh"
#include "vector3d.hh"
#include "atomic.h"

// field implementation

#define SMALL 0.000001


// Global parameters for redering the fields

double LJfield_maxradius=12.0; // 6-12 potential field
double Desol_maxradius=6.0; // from measure definition
double Elec_maxradius=12.0; // reasonable
double Shape_rho=8.0; // core penalty
double Shape_surface=3.4; // surface radius
double Shape_maxradius=3.4+3.0; // surface plus maximal vdw radius


// ***************************************************************
// GridSet definition
// ***************************************************************

GridSet::GridSet(double iscale,int isize) {
    init(iscale,isize);
}

GridSet::GridSet(void) {
    size=0;
    sizehalf=0;
    scale=0;
    sizesquared=0;
    sizecubed=0;
}

void GridSet::init(double iscale,int isize) {

    if (isize<2) throw "Gridset size must be atleast of size 2";
    if (iscale<=0) throw "Gridset scale must be larger than 0";
    size=isize;
    sizehalf=size/2;
    scale=iscale;
    sizesquared=size*size;
    sizecubed=sizesquared*size;
}

void GridSet::getclippedbox(const Vector3D center,
                            const double pixelwidth,
                            int min[3], int max[3]) {
    Vector3D scaledcenter;

    scaledcenter=center/scale;

    // calculate ranges in pixel space

    min[0]=scaledcenter[0]-pixelwidth;
    if (min[0]<0) min[0]=0;
    min[1]=scaledcenter[1]-pixelwidth;
    if (min[1]<0) min[1]=0;
    min[2]=scaledcenter[2]-pixelwidth;
    if (min[2]<0) min[2]=0;

    max[0]=scaledcenter[0]+pixelwidth+1;
    if (max[0]>size) max[0]=size;
    max[1]=scaledcenter[1]+pixelwidth+1;
    if (max[1]>size) max[1]=size;
    max[2]=scaledcenter[2]+pixelwidth+1;
    if (max[2]>size) max[2]=size;

}

// ***************************************************************
// Receptor grid
// ***************************************************************


ReceptorGridSet::ReceptorGridSet(double iscale,int isize) {
    GridSet::init(iscale,isize);
    init(iscale,isize);
}

void ReceptorGridSet::init(double iscale,int isize) {

    if (size==isize && scale==iscale) {
        clear();
        return;
    }
```

```
    GridSet::init(iscale,isize);

#ifdef DO_LJfield
    LJfield.init(scale,size); // 6-12 potential field
#endif

#ifdef DO_Desol
    DesolR.init(scale,size); // desolvation effects field
#endif

#ifdef DO_Elec
    Elecfield.init(scale,size); // electrical field
#endif

#ifdef DO_Shape
    ShapeR.init(scale,size); // receptor part of shape
#endif

}

void ReceptorGridSet::save(char *filename) {
    char fname[1024];

#ifdef DO_LJfield
    sprintf(fname,"%s.LJfield.grid",filename);
    LJfield.save(fname);
#endif

#ifdef DO_Desol
    sprintf(fname,"%s.DesolR.grid",filename);
    DesolR.save(fname);
#endif

#ifdef DO_Elec
    sprintf(fname,"%s.Elecfield.grid",filename);
    Elecfield.save(fname);
#endif

#ifdef DO_Shape
    sprintf(fname,"%s.ShapeR.grid",filename);
    ShapeR.save(fname);
#endif
}

void ReceptorGridSet::load(char *filename) {
    char fname[1024];

#ifdef DO_LJfield
    sprintf(fname,"%s.LJfield.grid",filename);
    LJfield.load(fname);
#endif

#ifdef DO_Desol
    sprintf(fname,"%s.DesolR.grid",filename);
    DesolR.load(fname);
#endif

#ifdef DO_Elec
    sprintf(fname,"%s.Elecfield.grid",filename);
    Elecfield.load(fname);
#endif

#ifdef DO_Shape
    sprintf(fname,"%s.ShapeR.grid",filename);
    ShapeR.load(fname);
#endif
}

void ReceptorGridSet::ifft() {
#ifdef DO_LJfield
    LJfield.ifft();
#endif
#ifdef DO_Desol
    DesolR.ifft();
#endif
#ifdef DO_Elec
    Elecfield.ifft();
#endif
#ifdef DO_Shape
    ShapeR.ifft();
#endif
}

void ReceptorGridSet::fft() {
#ifdef DO_LJfield
    LJfield.fft();
#endif
#ifdef DO_Desol
    DesolR.fft();
#endif
#ifdef DO_Elec
    Elecfield.fft();
#endif
#ifdef DO_Shape
    ShapeR.fft();
#endif
}

void ReceptorGridSet::clear( void ) {

#ifdef DO_LJfield
    LJfield.clear(); // 6-12 potential field
#endif
#ifdef DO_Desol
    DesolR.clear(); // desolvation effects
#endif
#ifdef DO_Elec
    Elecfield.clear(); // electrical field
#endif
#ifdef DO_Shape
    ShapeR.clear(); // receptor part of shape
#endif
}


// **************************************************
// method: ReceptorGridSet::rotate
// purpose:
// notes:
// Render a rotated/translated source grid with a field onto this grid.
// The grids are assumed not to have edge effects, fields in this
// grid outside the rotated grid (src) have value 0.0
// Grids are assumed to be centered at (sizehalf,sizehalf,sizehalf)
// extending sizehalf in each direction
// Merging is done by averaging on 'src' grid cells with a grid cell
// radius.
// Handles differences in scale and size

void ReceptorGridSet::rotate(const ReceptorGridSet &src,
                             const Vector3D &srczv,
                             const Vector3D &srcyv) {

    Vector3D dstcx; // grid corner x in destination grid
    Vector3D dstcy; // grid corner y in destination grid
    Vector3D dstcz; // grid corner z in destination grid

    Vector3D srcxv;

    Vector3D dstxv;
    Vector3D dstyv;
    Vector3D dstzv;

    int srcx,srcy,srcz;
    int dstx,dsty,dstz;
    Vector3D unitx(1.0,0.0,0.0);
    Vector3D unity(0.0,1.0,0.0);
    Vector3D unitz(0.0,0.0,1.0);
    Vector3D zero(0.0,0.0,0.0);
    Vector3D offset=zero; // we dont have offsets here
    Vector3D dstgridoffset;
    Matrix4D mat;
    int dstidx;
    int srcidx;
    double rescale;
    int srccellwidth;
    int srccellwb;
    int srccellwe;

    Vector3D srcp; // source point

    // The rescale tells how much longer/shorter a step in the dst grid
    // is compared to the src grid
    rescale=scale/src.scale;
    srccellwidth=floor(rescale/2.0);
    srccellwb=floor(rescale/3.0);
    srccellwe=ceil(rescale/3.0);

    // calculate xvector
    srcxv=srczv*srcyv;
    srcxv.normalize();

    // calculate destination unit vectors,
    // at zero, look along z vector, y is up
    mat.viewat(zero,srcxv,srcyv);
    dstxv=(unitx*mat)*rescale;
    dstyv=(unity*mat)*rescale;
    dstzv=(unitz*mat)*rescale;

    // offset is in the scale and space of the destination
```

```
    // and need to be transformed into the gridspace of the source
    dstgridoffset=((-offset/scale)*mat)*rescale;

    // calculate dst corner vectors
    dstcx=-dstxv*sizehalf;
    dstcy=-dstyv*sizehalf;
    dstcz=-dstzv*sizehalf;

#ifdef DEBUG
    cerr<<"true offset of source;"<<offset<<endl;
    cerr<<"destination grid offset of source;"<<offset/scale<<endl;
    cerr<<"source grid offset of destination"<<dstgridoffset<<endl;
    cerr<<"source system;"<<endl<<srcxv<<endl<<srcyv<<endl<<srczv<<endl;
    cerr<<"rescale;"<<rescale<<endl;
    cerr<<"srccellwidth;"<<-srccellwb<<".."<<srccellwe<<endl;
    cerr<<"destination system;"<<endl<<dstxv<<endl<<dstyv<<endl<<dstzv<<endl;
    cerr<<"destination corner;"<<endl<<dstcx<<endl<<dstcy<<endl<<dstcz<<endl;
    cerr<<"destination corner;"<<endl<<dstcx+dstcy+dstcz<<endl;
#endif

    // loop over destination
    for (dstidx=0,dstx=0;dstx<size;dstx++){
        for (dsty=0;dsty<size;dsty++){
            srcp=dstcx+dstcy+dstcz+dstgridoffset;
            for (dstz=0;dstz<size;dstz++,dstidx++){
                srcp+=dstzv; // step along zaxis
                if (srcp[0]<-src.sizehalf) continue;
                if (srcp[0]>=src.sizehalf) continue;
                if (srcp[1]<-src.sizehalf) continue;
                if (srcp[1]>=src.sizehalf) continue;
                if (srcp[2]<-src.sizehalf) continue;
                if (srcp[2]>=src.sizehalf) continue;

                srcx=int(srcp[0])+src.sizehalf;
                srcy=int(srcp[1])+src.sizehalf;
                srcz=int(srcp[2])+src.sizehalf;

                if (srccellwidth>=1) {
                    int count=0;
                    int i,j,k;
#ifdef DO_LJfield
                    Complex sumLJ;
                    sumLJ.re=0.0;
                    sumLJ.im=0.0;
#endif
#ifdef DO_Desol
                    double sumDesol;
                    sumDesol=0.0;
#endif
#ifdef DO_Elec
                    Complex sumElecfield; // electrical field
                    sumElecfield.re=0.0;
                    sumElecfield.im=0.0;
#endif
#ifdef DO_Shape
                    Complex sumShapeR;
                    sumShapeR.re=0.0;
                    sumShapeR.im=0.0;
#endif

                    // anitaliasing if compressed
                    for (i=srcx-srccellwb;i<srcx+srccellwe;i++){
                        if (i<0) continue;
                        if (i>src.size) continue;
                        for (j=srcy-srccellwb;j<srcy+srccellwe;j++){
                            if (j<0) continue;
                            if (j>src.size) continue;
                            for (k=srcz-srccellwb;k<srcz+srccellwe;k++){
                                if (k<0) continue;
                                if (k>src.size) continue;

                                srcidx=src.getidx(i,j,k);

#ifdef DO_LJfield
                                sumLJ=sumLJ+src.LJfield(srcidx);
#endif
#ifdef DO_Desol
                                sumDesol=sumDesol+src.DesolR(srcidx).re;
#endif
#ifdef DO_Elec
                                sumElecfield=sumElecfield+src.Elecfield(srcidx);
#endif
#ifdef DO_Shape
                                sumShapeR=sumShapeR+src.ShapeR(srcidx);
#endif
                                count++;
                            }
                        }
                    }
                    if (count==0) count=1;
#ifdef DO_LJfield
                    LJfield(dstidx).re=sumLJ.re/double(count);
                    LJfield(dstidx).im=sumLJ.im/double(count);
#endif
#ifdef DO_Desol
                    DesolR(dstidx).re=sumDesol/double(count);
#endif
#ifdef DO_Elec
                    Elecfield(dstidx).re=sumElecfield.re/double(count);
                    Elecfield(dstidx).im=sumElecfield.im/double(count);
#endif
#ifdef DO_Shape
                    ShapeR(dstidx).re=sumShapeR.re/double(count);
                    ShapeR(dstidx).im=sumShapeR.im/double(count);
#endif
                }else{
                    // just one value
                    srcidx=src.getidx(srcx,srcy,srcz);
                    // no antialiasing
#ifdef DO_LJfield
                    LJfield(dstidx)=src.LJfield(srcidx);
#endif
#ifdef DO_Desol
                    DesolR(dstidx).re=src.DesolR(srcidx).re;
#endif
#ifdef DO_Elec
                    Elecfield(dstidx)=src.Elecfield(srcidx);
#endif
#ifdef DO_Shape
                    ShapeR(dstidx)=src.ShapeR(srcidx);
#endif
                }

            }
            dstcy+=dstyv; // step along yaxis
        }
        dstcy-=dstyv*size;
        dstcx+=dstxv; // step along xaxis
    }

    // Render a rotated/translated source grid with _delta_
    // function onto this grid.
    // As delta functions occur in single points in the source and
    // may not be smoothed. In fact they must occur as a sum in the
    // resulting rotated grid. Therefore this rotation loops over
    // the source grid points instead of the destination
    // grid points. Important of the delta grid is scaled up.
    // The grids are assumed not to have edge effects, fields in this
    // grid outside the rotated grid (src) have value 0.0
    // Grids are assumed to be centered at (sizehalf,sizehalf,sizehalf)
    // extending sizehalf in each direction
    // Merging is done by averaging on 'src' grid cells with a grid cell
    // radius.
    // Handles difference in scale and size
    // srcx,srcy,srcz: source grid normal vectors in destination space.

    Vector3D srccx; // grid corner x in destination grid
    Vector3D srccy; // grid corner y in destination grid
    Vector3D srccz; // grid corner z in destination grid
    Vector3D srcxvs;
    Vector3D srcyvs;
    Vector3D srczvs;
    Vector3D srcgridoffset;
    Vector3D dstp; // destination point

    rescale=scale/src.scale;
    srcgridoffset=offset/scale;

    // calculate corner of source
    srcxvs=srcxv/rescale;
    srcyvs=srcyv/rescale;
    srczvs=srczv/rescale;

    srccx=-srcxvs*src.sizehalf;
    srccy=-srcyvs*src.sizehalf;
    srccz=-srczvs*src.sizehalf;

#ifdef DEBUG
    cerr<<"rescale;"<<rescale<<endl;
    cerr<<"destination offset of source;"<<offset<<endl;
    cerr<<"destination grid offset of source;"<<srcgridoffset<<endl;
    cerr<<"Source system;"<<endl<<srcxv<<endl<<srcyv<<endl<<srczv<<endl;
    cerr<<"Source corner;"<<endl<<srccx<<endl<<srccy<<endl<<srccz<<endl;
    cerr<<"Source corner;"<<endl<<srccx+srccy+srccz<<endl;
#endif

    // loop over source
    for (srcidx=0,srcx=0;srcx<src.size;srcx++){
        for (srcy=0;srcy<src.size;srcy++){
```

```
            dstp=srccx+srccy+srccz+srcgridoffset;
            for (srcz=0;srcz<src.size;srcz++,srcidx++){
                dstp+=srczvs; // step along zaxis
                if (dstp[0]<-sizehalf) continue;
                if (dstp[0]>=sizehalf) continue;
                if (dstp[1]<-sizehalf) continue;
                if (dstp[1]>=sizehalf) continue;
                if (dstp[2]<-sizehalf) continue;
                if (dstp[2]>=sizehalf) continue;
                dstx=int(dstp[0])+sizehalf;
                dsty=int(dstp[1])+sizehalf;
                dstz=int(dstp[2])+sizehalf;
                dstidx=getidx(dstx,dsty,dstz);

                // no antialiasing but sum
#ifdef DO_Desol
                DesolR(dstidx).im+=src.DesolR(srcidx).im;
#endif
            }
            srccy+=srcyvs; // step along yaxis
        }
        srccy-=srcyvs*src.size;
        srccx+=srcxvs; // step along xaxis
    }

}


// ****************************************************************
// Ligand grid
// ****************************************************************

LigandGridSet::LigandGridSet(double iscale,int isize) {
    GridSet::init(iscale,isize);
    init(iscale,isize);
}

void LigandGridSet::init(double iscale,int isize) {
    GridSet::init(iscale,isize);

#ifdef DO_Desol
    DesolL.init(scale,size); // desolvation effects density
#endif
#ifdef DO_Elec
    Elecd.init(scale,size); // electrical density
#endif
#ifdef DO_Shape
    ShapeL.init(scale,size); // ligand part of shape
#endif

    Density.init(scale,size); // density (delta function)
}

void LigandGridSet::save(char *filename) {
    char fname[1024];

#ifdef DO_Desol
    sprintf(fname,"%s.DesolL.grid",filename);
    DesolL.save(fname);
#endif
#ifdef DO_Elec
    sprintf(fname,"%s.Elecd.grid",filename);
    Elecd.save(fname);
#endif
#ifdef DO_Shape
    sprintf(fname,"%s.ShapeL.grid",filename);
    ShapeL.save(fname);
#endif
    sprintf(fname,"%s.Density.grid",filename);
    Density.save(fname);
}

void LigandGridSet::load(char *filename) {
    char fname[1024];

#ifdef DO_Desol
    sprintf(fname,"%s.DesolL.grid",filename);
    DesolL.load(fname);
#endif
#ifdef DO_Elec
    sprintf(fname,"%s.Elecd.grid",filename);
    Elecd.load(fname);
#endif
#ifdef DO_Shape
    sprintf(fname,"%s.ShapeL.grid",filename);
    ShapeL.load(fname);
#endif
    sprintf(fname,"%s.Density.grid",filename);
    Density.load(fname);
}

void LigandGridSet::clear( void ) {
#ifdef DO_Desol
    DesolL.clear(); // desolvation effects
#endif
#ifdef DO_Elec
    Elecd.clear(); // electrical density
#endif
#ifdef DO_Shape
    ShapeL.clear(); // ligand part of shape
#endif
    Density.clear(); // density (delta function)
}

void LigandGridSet::ifft() {
#ifdef DO_Desol
    DesolL.ifft();
#endif
#ifdef DO_Elec
    Elecd.ifft();
#endif
#ifdef DO_Shape
    ShapeL.ifft();
#endif
    Density.ifft();
}

void LigandGridSet::fft() {
#ifdef DO_Desol
    DesolL.fft();
#endif
#ifdef DO_Elec
    Elecd.fft();
#endif
#ifdef DO_Shape
    ShapeL.fft();
#endif
    Density.fft();
}

void LigandGridSet::rotate(const LigandGridSet &src,
                                 const Vector3D &srczv,
                                 const Vector3D &srcyv) {

    // First the field functions are rendered

    // Render a rotated/translated source grid with a field onto this grid.
    // The grids are assumed not to have edge effects, fields in this
    // grid outside the rotated grid (src) have value 0.0
    // Grids are assumed to be centered at (sizehalf,sizehalf,sizehalf)
    // extending sizehalf in each direction
    // Merging is done by averaging on 'src' grid cells with a grid cell
    // radius.
    // Handles differences in scale and size

    Vector3D dstcx; // grid corner x in destination grid
    Vector3D dstcy; // grid corner y in destination grid
    Vector3D dstcz; // grid corner z in destination grid

    Vector3D srcxv;

    Vector3D dstxv;
    Vector3D dstyv;
    Vector3D dstzv;

    int srcx,srcy,srcz;
    int dstx,dsty,dstz;
    Vector3D unitx(1.0,0.0,0.0);
    Vector3D unity(0.0,1.0,0.0);
    Vector3D unitz(0.0,0.0,1.0);
    Vector3D zero(0.0,0.0,0.0);
    Vector3D offset=zero; // we dont have offsets here
    Vector3D dstgridoffset;
    Matrix4D mat;
    int dstidx;
    int srcidx;
    double rescale;
    int srccellwidth;
    int srccellwb;
    int srccellwe;

    Vector3D srcp; // source point

    // The rescale tells how much longer/shorter a step in the dst grid
    // is compared to the src grid
    rescale=scale/src.scale;
    srccellwidth=floor(rescale/2.0);
    srccellwb=floor(rescale/3.0);
    srccellwe=ceil(rescale/3.0);
```

```
    // calculate xvector
    srcxv=srczv*srcyv;
    srcxv.normalize();

    // calculate destination unit vectors, at zero,
    // look along z vector, y is up
    mat.viewat(zero,srcxv,srcyv);
    dstxv=(unitx*mat)*rescale;
    dstyv=(unity*mat)*rescale;
    dstzv=(unitz*mat)*rescale;

    // offset is in the scale and space of the destination
    // and need to be transformed into the gridspace of the source
    dstgridoffset=((-offset/scale)*mat)*rescale;

    // calculate dst corner vectors
    dstcx=-dstxv*sizehalf;
    dstcy=-dstyv*sizehalf;
    dstcz=-dstzv*sizehalf;

#ifdef DEBUG
    cerr≪"true offset of source;"≪offset≪endl;
    cerr≪"destination grid offset of source;"≪offset/scale≪endl;
    cerr≪"source grid offset of destination"≪dstgridoffset≪endl;
    cerr≪"source system;"≪endl≪srcxv≪endl≪srcyv≪endl≪srczv≪endl;
    cerr≪"rescale;"≪rescale≪endl;
    cerr≪"srccellwidth;"≪-srccellwb≪".."≪srccellwe≪endl;
    cerr≪"destination system;"≪endl≪dstxv≪endl≪dstyv≪endl≪dstzv≪endl;
    cerr≪"destination corner;"≪endl≪dstcx≪endl≪dstcy≪endl≪dstcz≪endl;
    cerr≪"destination corner;"≪endl≪dstcx+dstcy+dstcz≪endl;
#endif

    // Render a rotated/translated source grid with _delta_
    // function onto this grid.
    // As delta functions occur in single points in the source and
    // may not be smoothed. In fact they must occur as a sum in the
    // resulting rotated grid. Therefore this rotation loops over
    // the source grid points instead of the destination
    // grid points. Important of the delta grid is scaled up.
    // The grids are assumed not to have edge effects, fields in this
    // grid outside the rotated grid (src) have value 0.0
    // Grids are assumed to be centered at (sizehalf,sizehalf,sizehalf)
    // extending sizehalf in each direction
    // Merging is done by averaging on 'src' grid cells with a grid cell
    // radius.
    // Handles difference in scale and size
    // srcx,srcy,srcz: source grid normal vectors in destination space.

    // only shape is a field in the ligand
    // loop over destination
    for (dstidx=0,dstx=0;dstx<size;dstx++){
        for (dsty=0;dsty<size;dsty++){
            srcp=dstcx+dstcy+dstcz+dstgridoffset;
            for (dstz=0;dstz<size;dstz++,dstidx++){
                srcp+=dstzv; // step along zaxis
                if (srcp[0]<-src.sizehalf) continue;
                if (srcp[0]≥src.sizehalf) continue;
                if (srcp[1]<-src.sizehalf) continue;
                if (srcp[1]≥src.sizehalf) continue;
                if (srcp[2]<-src.sizehalf) continue;
                if (srcp[2]≥src.sizehalf) continue;

                srcx=int(srcp[0])+src.sizehalf;
                srcy=int(srcp[1])+src.sizehalf;
                srcz=int(srcp[2])+src.sizehalf;

                if (srccellwidth≥1) {
                    int count=0;
                    int i,j,k;
#ifdef DO_Shape
                    Complex sumShapeL;
                    sumShapeL.re=0;
                    sumShapeL.im=0;
#endif


                    // anitaliasing if compressed
                    for (i=srcx-srccellwb;i<srcx+srccellwe;i++){
                        if (i<0) continue;
                        if (i>src.size) continue;
                        for (j=srcy-srccellwb;j<srcy+srccellwe;j++){
                            if (j<0) continue;
                            if (j>src.size) continue;
                            for (k=srcz-srccellwb;k<srcz+srccellwe;k++){
                                if (k<0) continue;
                                if (k>src.size) continue;

                                srcidx=src.getidx(i,j,k);

#ifdef DO_Shape
                                sumShapeL=sumShapeL+src.ShapeL(srcidx);
#endif

                                count++;
                            }
                        }
                    }

                    if (count==0) count=1;

#ifdef DO_Shape
                    ShapeL(dstidx).re=sumShapeL.re/double(count);
                    ShapeL(dstidx).im=sumShapeL.im/double(count);
#endif
                }else{
                    // just one value
                    srcidx=src.getidx(srcx,srcy,srcz);
#ifdef DO_Shape
                    // no antialiasing
                    ShapeL(dstidx)=src.ShapeL(srcidx);
#endif
                }

            }
            dstcy+=dstyv; // step along yaxis
        }
        dstcy-=dstyv*size;
        dstcx+=dstxv; // step along xaxis
    }

    Vector3D srccx; // grid corner x in destination grid
    Vector3D srccy; // grid corner y in destination grid
    Vector3D srccz; // grid corner z in destination grid
    Vector3D srcxvs;
    Vector3D srcyvs;
    Vector3D srczvs;
    Vector3D srcgridoffset;
    Vector3D dstp; // destination point

    rescale=scale/src.scale;
    srcgridoffset=offset/scale;

    // calculate corner of source
    srcxvs=srcxv/rescale;
    srcyvs=srcyv/rescale;
    srczvs=srczv/rescale;

    srccx=-srcxvs*src.sizehalf;
    srccy=-srcyvs*src.sizehalf;
    srccz=-srczvs*src.sizehalf;

#ifdef DEBUG
    cerr≪"rescale;"≪rescale≪endl;
    cerr≪"destination offset of source;"≪offset≪endl;
    cerr≪"destination grid offset of source;"≪srcgridoffset≪endl;
    cerr≪"Source system;"≪endl≪srcxv≪endl≪srcyv≪endl≪srczv≪endl;
    cerr≪"Source corner;"≪endl≪srccx≪endl≪srccy≪endl≪srccz≪endl;
    cerr≪"Source corner;"≪endl≪srccx+srccy+srccz≪endl;
#endif

    // loop over source
    for (srcidx=0,srcx=0;srcx<src.size;srcx++){
        for (srcy=0;srcy<src.size;srcy++){
            dstp=srccx+srccy+srccz+srcgridoffset;
            for (srcz=0;srcz<src.size;srcz++,srcidx++){
                dstp+=srczvs; // step along zaxis
                if (dstp[0]<-sizehalf) continue;
                if (dstp[0]≥sizehalf) continue;
                if (dstp[1]<-sizehalf) continue;
                if (dstp[1]≥sizehalf) continue;
                if (dstp[2]<-sizehalf) continue;
                if (dstp[2]≥sizehalf) continue;
                dstx=int(dstp[0])+sizehalf;
                dsty=int(dstp[1])+sizehalf;
                dstz=int(dstp[2])+sizehalf;
                dstidx=getidx(dstx,dsty,dstz);

                // no antialiasing but sum
#ifdef DO_Elec
                Elecd(dstidx)=Elecd(dstidx)+src.Elecd(srcidx);
#endif

#ifdef DO_Desol
                DesolL(dstidx).im+=src.DesolL(srcidx).im;
#endif

                Density(dstidx)=Density(dstidx)+src.Density(srcidx);

            }
            srccy+=srcyvs; // step along yaxis
```

```
        }
        srccy-=srcyvs*src.size;
        srccx+=srcxvs; // step along xaxis
    }
}

// merging function Not finished yet
void LigandGridSet::merge(const LigandGridSet &src) {
    int i;

#ifdef UNFINISHED
    for (i=0;i<sizecubed;i++){
#ifdef DO_Shape
        // shape; the imaginary core penalties should be the minimum
        if (src.Shaped(i).re>Shaped(i).re) Shaped(i).re=src.Shaped(i).re;
        if (src.Shaped(i).im<Shaped(i).im) Shaped(i).im=src.Shaped(i).im;
#endif
#ifdef DO_Desol
        // desolvent; the imaginary density should be maximized
        if (src.Desold(i).re>Desold(i).re) Desold(i).re=src.Desold(i).re;
        if (src.Desold(i).im>Desold(i).im) Desold(i).im=src.Desold(i).im;
#endif
#ifdef DO_Elec
        // electrical bound;
        // mostpositive: mp
        // fieldnegative: fn
        // mostnegative: mn
        // fieldpositive: fp
        // leastnegative: ln
        // leastpositive: lp
        //
        // bound= mp*fn + mn*fp - ( ln*fn + lp*fp )
        //
        // f = ( fn, fp )
        // d1 = ( mp, mn )
        // d2 = ( lp, ln )
        //
        // f*d1 = ( fn*mp - fp*mn ), ( fn*mn + fp*mp )
        // = ( strongestattraction, strongestrepulsion )
        //
        // f*d2 = ( fn*lp - fp*ln ), ( fn*ln + fp*lp )
        // ( weakestattraction, weakestrepulsion )
        // score= bestattraction - worstrepulsion
        //
        // electrical; the imaginary point charges should me
        // maximized for best interaction
        if (src.Elecdm(i).re>Elecdm(i).re) Elecdm(i).re=src.Elecdm(i).re;
        if (src.Elecdm(i).im>Elecdm(i).re) Elecdm(i).re=src.Elecdm(i).re;

        if (src.Elecdl(i).im>Elecd(i).im) Elecd(i).im=src.Elecd(i).im;
#endif
#ifdef DO_JLfield
        if (src.Density(i).re>Density(i).re) Density(i).re=src.Density(i).re;
#endif
    }

#endif
}
```

rotationspace.hh

```
// **************************************************
// File: rotationspace.hh
// purpose: rotation space subdivided class structure
//
// author: Hans-Henrik Staerfeldt
// notes:
//
// The rotation space data structure is a tree that allows for the
// construction and subdivision of rotational space. The space is
// initially divided into 160 segments, subdividing 2 rotational
// dimentions as the triangulations of a sphere (20 segments) and
// for each segment, the third rotational dimention is initially
// divided into 8 segments (optimal is 7.92 to match the other).
//
// Subdividing a segment will cause the first two dimentions to be
// subdivided into 4 segments (by triangulating the triangular segment),
// and the lst dimention is subdivided into two segments, making
// 8 children.
//

#ifndef ROTATIONSPACE_HH
#define ROTATIONSPACE_HH

#include <vector>
#include "vector3d.hh"

using namespace std;

#define ROTATIONSEGMENT_CHILDREN 8
#define ROTATIONSPACE_CHILDREN 160

// **************************************************
// template class: Rotationsegment
// purpose: segment of the subdivided rotationspace
// notes:

template <class Data> class Rotationsegment {

 private:
    double sphi[3]; // sphere angles, (corners) 3 phi angles
    double stheta[3]; // sphere angles, (corners) 3 theta angles
    double spsi[2]; // psi angles, (range) 2 psi angles

    double phi; // sphere angle (center)
    double theta; // sphere angle (center)
    double psi; // psi angle (center)

    int nchildren;
    class Rotationsegment *child[ROTATIONSEGMENT_CHILDREN];

 public:
    Data *data;
    Rotationsegment( void );
    Rotationsegment(const double phi[3],
                    const double theta[3],
                    const double psi[2]);

    ~Rotationsegment( void );
    void subdivide(const int debth);
    void makeleaf(void);
    int isleaf(void);
    class Rotationsegment <Data> *getchild(const int sn);
    void render( int debth ); // write 3D model of segmentation
    void getvectors(double phi[3],
                    double theta[3],
                    double psi[2]);
    void getvector(double ophi,
                   double otheta,
                   double opsi);
    void setvectors(const double phi[3],
                    const double theta[3],
                    const double psi[2]);
    void setvector(double ophi,
                   double otheta,
                   double opsi);
    int size( void ); // get number of leaf points
    void collectleafs( vector <Vector3D> &angles );
    void collectsegments( vector <Vector3D> &angles );
};

// **************************************************
// template class: Rotationspace
// purpose: rotationspace initial subdivision class
// notes:

template <class Data> class Rotationspace {
 private:
    class Rotationsegment <Data> *child[ROTATIONSPACE_CHILDREN];
    void setdebth( const int debth );
    // subdivide space by debth
 public:
    int size( void ); // number of points in space
    void render( void ); // write 3D model of segmentation
    // subdivide space by reference points
    class Rotationsegment <Data> *getchild(const int sn);
    void collectleafs( vector <Vector3D> &angles );
    void collectsegments( vector <Vector3D> &angles );
    // subdivide space by debth
    void subdivide( const int maxdebth );
    static void getmatrix(const double phi,
                          const double theta,
                          const double psi,
                          Matrix4D &rot);
    Rotationspace( void );
    ~Rotationspace( void );
};

#endif
```

rotationspace.cc

```
// **************************************************
// File: rotationspace.cc
// purpose: implementation of rotation space subdivision
// and traversal
// author: Hans-Henrik Staerfeldt
// notes:
```

```
#include "rotationspace.hh"
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include "vector3d.hh"


static double average(double &r1, double &r2) {
    if (r2>r1) {
        if (r2-r1>PI)
          r2-=PI2;
    }else{
        if (r1-r2>PI)
          r1-=PI2;
    }
    return (r1+r2)/2.0;
}

static void average(const double phi1,
                    const double phi2,
                    const double phi3,
                    const double theta1,
                    const double theta2,
                    const double theta3,
                    double &phi,
                    double &theta) {
    Vector3D v1;
    Vector3D v2;
    Vector3D v3;
    Vector3D v;
    double dummy;

    v1.setspherical(phi1,theta1,1);
    v2.setspherical(phi2,theta2,1);
    v3.setspherical(phi3,theta3,1);

    v=(v1+v2+v3)/3.0;
    v.getspherical(phi,theta,dummy);
}

static void average(const double phi1,
                    const double phi2,
                    const double theta1,
                    const double theta2,
                    double &phi,
                    double &theta) {
    Vector3D v1;
    Vector3D v2;
    Vector3D v;
    double dummy;

    v1.setspherical(phi1,theta1,1);
    v2.setspherical(phi2,theta2,1);

    v=(v1+v2)/2.0;
    v.getspherical(phi,theta,dummy);
}


// #define DEBUG

template <class Data> Rotationsegment <Data>::Rotationsegment(void) {
    int i;

#ifdef DEBUG2
    printf("Construct Segment\n");
#endif

    // initialize
    phi=sphi[0]=sphi[1]=0;
    theta=stheta[0]=stheta[1]=stheta[2]=0;
    psi=spsi[0]=spsi[1]=spsi[2]=0;
    nchildren=0;
    for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++)
      child[i]=NULL;
    data=NULL;
}

template <class Data>
void Rotationsegment <Data>::getvectors(double ophi[3],
                                        double otheta[3],
                                        double opsi[2]) {
    ophi[0]=sphi[0];
    ophi[1]=sphi[1];
    ophi[2]=sphi[2];
    otheta[0]=stheta[0];
    otheta[1]=stheta[1];
    otheta[2]=stheta[2];
    opsi[0]=spsi[0];
```

```
    opsi[1]=spsi[1];
}

template <class Data>
void Rotationsegment <Data>::getvector(double ophi,
                                       double otheta,
                                       double opsi) {
    ophi=phi;
    otheta=theta;
    opsi=psi;
}

template <class Data>
void Rotationsegment <Data>::setvectors(const double iphi[3],
                                        const double itheta[3],
                                        const double ipsi[2]) {
    sphi[0]=iphi[0];
    sphi[1]=iphi[1];
    sphi[2]=iphi[2];
    stheta[0]=itheta[0];
    stheta[1]=itheta[1];
    stheta[2]=itheta[2];
    spsi[0]=ipsi[0];
    spsi[1]=ipsi[1];
}

template <class Data>
void Rotationsegment <Data>::setvector(double iphi,
                                       double itheta,
                                       double ipsi) {
    phi=iphi;
    theta=itheta;
    psi=ipsi;
}


template <class Data>
Rotationsegment <Data>::Rotationsegment(const double iphi[3],
                                        const double itheta[3],
                                        const double ipsi[2]){
    int i;
#ifdef DEBUG2
    printf("Construct Segment\n");
#endif
    sphi[0]=iphi[0];
    sphi[1]=iphi[1];
    sphi[2]=iphi[2];
    stheta[0]=itheta[0];
    stheta[1]=itheta[1];
    stheta[2]=itheta[2];
    spsi[0]=ipsi[0];
    spsi[1]=ipsi[1];

    average(iphi[0],iphi[1],iphi[2],
            itheta[0],itheta[1],itheta[2],
            phi,theta);

    nchildren=0;
    for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++)
      child[i]=NULL;
}

template <class Data> int Rotationsegment <Data>::isleaf( void ){
    if (nchildren==0) return 1;
    return 0;
}

template <class Data> void Rotationsegment <Data>::makeleaf( void ){
    int i;

    nchildren=0;
    for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++)
      if (child[i]≠NULL) {
          delete child[i];
          child[i]=NULL;
      }
}

template <class Data> int Rotationsegment <Data>::size( void ){
    int pointcnt=0;
    int i;

    if (nchildren==0) return 1;
    for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++)
      if (child[i]≠NULL)
        pointcnt+=child[i]→size();
    return pointcnt;
}

template <class Data>
```

```
void Rotationsegment <Data>::collectleafs( vector <Vector3D> &angles ) {
    Vector3D angle;
    int i;
    if (isleaf()) {
        angle.set(phi,theta,psi);
        angles.push_back(angle);
    }else{
        for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++) {
            if (child[i]≠NULL)
                child[i]→collectleafs(angles);
        }
    }
}

template <class Data>
void Rotationsegment <Data>::collectsegments( vector <Vector3D> &angles ) {
    Vector3D angle;
    int i;

    if (isleaf()) return;

    for (i=0;i<3;i++) {
        angle.set(sphi[i],stheta[i],spsi[0]);
        angles.push_back(angle);
        angle.set(sphi[i],stheta[i],spsi[1]);
        angles.push_back(angle);
    }

    for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++) {
        if (child[i]≠NULL)
            child[i]→collectsegments(angles);
    }
}


//
// Subdivide a rotationspace into 8 segments
//

template <class Data>
void Rotationsegment <Data>::subdivide(const int debth) {
    double iphi[3];
    double itheta[3];
    double ipsi[2];

#ifdef DEBUG
    printf("Subdividing segment\n");
#endif

    if (debth≤0) {
        makeleaf();
        return;
    }

    nchildren=0;

    // first segments with first psi-range
    ipsi[0]=spsi[0];
    ipsi[1]=(spsi[0]+spsi[1])/2.0;

    // first phi/theta subdivision

    iphi[0]=sphi[0];itheta[0]=stheta[0];
    average(sphi[1],sphi[0],stheta[1],stheta[0],iphi[1],itheta[1]);
    average(sphi[2],sphi[0],stheta[2],stheta[0],iphi[2],itheta[2]);
    child[nchildren]=new Rotationsegment(iphi,itheta,ipsi);
    child[nchildren++]→subdivide(debth-1);

    // second phi/theta subdivision
    average(sphi[2],sphi[0],stheta[2],stheta[0],iphi[0],itheta[0]);
    average(sphi[2],sphi[1],stheta[2],stheta[1],iphi[1],itheta[1]);
    iphi[2]=sphi[2];itheta[2]=stheta[2];
    child[nchildren]=new Rotationsegment(iphi,itheta,ipsi);
    child[nchildren++]→subdivide(debth-1);

    // third phi/theta subdivision
    average(sphi[1],sphi[0],stheta[1],stheta[0],iphi[0],itheta[0]);
    iphi[1]=sphi[1];itheta[1]=stheta[1];
    average(sphi[2],sphi[1],stheta[2],stheta[1],iphi[2],itheta[2]);
    child[nchildren]=new Rotationsegment(iphi,itheta,ipsi);
    child[nchildren++]→subdivide(debth-1);

    // fourth phi/theta subdivision
    average(sphi[2],sphi[0],stheta[2],stheta[0],iphi[0],itheta[0]);
    average(sphi[0],sphi[1],stheta[0],stheta[1],iphi[1],itheta[1]);
    average(sphi[2],sphi[1],stheta[2],stheta[1],iphi[2],itheta[2]);
    child[nchildren]=new Rotationsegment(iphi,itheta,ipsi);
    child[nchildren++]→subdivide(debth-1);

    // second segment with second psi-range
    ipsi[0]=(spsi[0]+spsi[1])/2.0;
    ipsi[1]=spsi[1];

    // first phi/theta subdivision

    iphi[0]=sphi[0];itheta[0]=stheta[0];
    average(sphi[1],sphi[0],stheta[1],stheta[0],iphi[1],itheta[1]);
    average(sphi[2],sphi[0],stheta[2],stheta[0],iphi[2],itheta[2]);
    child[nchildren]=new Rotationsegment(iphi,itheta,ipsi);
    child[nchildren++]→subdivide(debth-1);

    // second phi/theta subdivision
    average(sphi[2],sphi[0],stheta[2],stheta[0],iphi[0],itheta[0]);
    average(sphi[2],sphi[1],stheta[2],stheta[1],iphi[1],itheta[1]);
    iphi[2]=sphi[2];itheta[2]=stheta[2];
    child[nchildren]=new Rotationsegment(iphi,itheta,ipsi);
    child[nchildren++]→subdivide(debth-1);

    // third phi/theta subdivision
    average(sphi[1],sphi[0],stheta[1],stheta[0],iphi[0],itheta[0]);
    iphi[1]=sphi[1];itheta[1]=stheta[1];
    average(sphi[2],sphi[1],stheta[2],stheta[1],iphi[2],itheta[2]);
    child[nchildren]=new Rotationsegment(iphi,itheta,ipsi);
    child[nchildren++]→subdivide(debth-1);

    // fourth phi/theta subdivision
    average(sphi[2],sphi[0],stheta[2],stheta[0],iphi[0],itheta[0]);
    average(sphi[0],sphi[1],stheta[0],stheta[1],iphi[1],itheta[1]);
    average(sphi[2],sphi[1],stheta[2],stheta[1],iphi[2],itheta[2]);
    child[nchildren]=new Rotationsegment(iphi,itheta,ipsi);
    child[nchildren++]→subdivide(debth-1);

}

template <class Data> Rotationsegment <Data>::~Rotationsegment( void ) {
    int i;

#ifdef DEBUG2
    printf("Deconstruct segment\n");
#endif
    for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++) {
        if (child[i]≠NULL ) {
            delete child[i];
            child[i]=NULL;
        }
    }
    if (data) delete data;
}

template <class Data>
class Rotationsegment <Data> *Rotationsegment <Data> ::getchild(int sn) {
    if (sn<0 || sn≥ROTATIONSEGMENT_CHILDREN) return NULL;
    return child[sn];
}

template <class Data>
class Rotationsegment <Data> *Rotationspace <Data>::getchild(const int sn) {
    if (sn<0 || sn≥ROTATIONSPACE_CHILDREN) return NULL;
    return child[sn];
}

template <class Data> Rotationspace <Data>::~Rotationspace( void ) {
    int i;
#ifdef DEBUG2
    printf("Deconstruct space\n");
#endif
    for (i=0;i<ROTATIONSPACE_CHILDREN;i++) {
        if (child[i]≠NULL ) {
            delete child[i];
            child[i]=NULL;
        }
    }
}

template <class Data> int Rotationspace <Data>::size( void ) {
    int pointcnt=0;
    int i;

    for (i=0;i<ROTATIONSPACE_CHILDREN;i++) {
        if (child[i]≠NULL )
            pointcnt+=child[i]→size();
    }
    return pointcnt;
}

template <class Data> void Rotationsegment <Data>::render( int debth ){

    int i;

    if (isleaf()) {
```

```
        printf("L %f %f %f\n",
                phi,theta,psi);
        return;
    }

    printf("S %f %f %f , %f %f %f , %f %f\n",
            sphi[0], sphi[1], sphi[2],
            stheta[0], stheta[1], stheta[2],
            spsi[0],spsi[1]);

    for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++) {
        if (child[i]≠NULL )
          child[i]→render(debth+1);
    }

}

// render Rotation space
template <class Data> void Rotationspace <Data>::render( void ) {
    int i;
    printf("Rotationspace {\n");

    // traverse space and generate coordinates for leaves
    for (i=0;i<ROTATIONSPACE_CHILDREN;i++) {
        if (child[i]≠NULL ){
            child[i]→render(0);
        }
    }
    printf("}\n");
}

template <class Data>
void Rotationspace <Data>::getmatrix(const double phi,
                                     const double theta,
                                     const double psi,
                                     Matrix4D &rot) {
    static const Vector3D o(0.0,0.0,0.0);
    static const Vector3D vzo(0.0,0.0,1.0);
    static const Vector3D vyo(0.0,1.0,0.0);
    Vector3D vz;
    Vector3D vy;

    // transform pherical coordinate thi,theta,1.0 to z axis
    vz.setspherical(phi,theta,1.0);
    if (theta==0.0) {
        // it is along the _zaxis_, use yaxis as 'up'
        // rotate z axis psi radians around vz as 'up'
        rot.rotation(Quaternion(vz,psi));
        vy=vyo*rot;
        // use the rotated yaxis to build rotation
    }else{
        // its not along the zaxis, use zaxis as 'up'
        // rotate y axis psi radians around vz as 'up'
        rot.rotation(Quaternion(vz,psi));
        vy=vzo*rot;
        // use the rotated zaxis to build rotation
    }
    // use the rotated 'up' axis to build rotation
    rot.viewat(o,vz,vy);
}

template <class Data>
void Rotationspace <Data>::subdivide( const int debth ) {
    int i;

    if (debth==0) return;

#ifdef DEBUG
    printf("Subdivide space to %d debth\n",debth);
#endif
    for (i=0;i<ROTATIONSPACE_CHILDREN;i++) {
        if (child[i]≠NULL)
          child[i]→subdivide(debth-1);
    }
}

template <class Data>
void Rotationspace <Data>::collectleafs( vector <Vector3D> &angles ) {
    int i;
    for (i=0;i<ROTATIONSPACE_CHILDREN;i++) {
        if (child[i]≠NULL)
          child[i]→collectleafs(angles);
    }
}

template <class Data>
void Rotationspace <Data>::collectsegments( vector <Vector3D> &angles ) {
    int i;
    for (i=0;i<ROTATIONSPACE_CHILDREN;i++) {
        if (child[i]≠NULL)
            child[i]→collectsegments(angles);
    }
}

template <class Data> Rotationspace <Data>::Rotationspace( void ) {

    // unnormalized coordinates;
    // (+- 1, 0, +-t), (0, +-t, +-1), and (+-t, +-1, 0), t=0.61803398874

    // initial sectioning of w angle is
    // 2pi/sqrt(4pi/20)=7.91=8
    // to match the 20 sided sectioning of the u,v angles

    int i,j;
    int s;
    double ipsi[8][2];

    double phi[3]; // current angle
    double theta[3]; // current angle
    double psi[2]; // current angle

    // this defines the points of a isocaeder in spherical coordinates
    static double icos_coords[12][3]={
      {1, 0, 0.61803398874},
      {1, 0, -0.61803398874},
      {-1, 0, 0.61803398874},
      {-1, 0, -0.61803398874},
      {0, 0.61803398874, 1},
      {0, 0.61803398874, -1},
      {0, -0.61803398874, 1},
      {0, -0.61803398874, -1},
      {0.61803398874, 1, 0},
      {0.61803398874, -1, 0},
      {-0.61803398874, 1, 0},
      {-0.61803398874, -1, 0}
    };

    // this defines the surface of a isocaeder
    static const int icos_surface[20][3]={
      {2, 4, 10}, {2, 4, 6}, {2, 6, 11}, {2, 3, 11}, {2, 3, 10},
      {3, 7, 11}, {3, 5, 10}, {10, 8, 4}, {4, 0, 6}, {6, 9, 11},
      {7, 9, 11}, {10, 5, 8}, {0, 6, 9}, {8, 4, 0}, {7, 3, 5},
      {1, 0, 9}, {9, 1, 7}, {1, 5, 7}, {1, 5, 8}, {1, 0, 8}
    };
    Vector3D v;
    double dummy;

    // calculate all inital psi angles.
    for (j=0;j<8;j++) {
        ipsi[j][0]=j*2.0*PI/8.0;
        ipsi[j][1]=(j+1)*2.0*PI/8.0;
    }

    // generate all 160 segments
    for ( s=0,i=0;i<20;i++ ){

        for (j=0;j<3;j++) {
            Vector3D v2;
            v.set(icos_coords[icos_surface[i][j]][0],
                  icos_coords[icos_surface[i][j]][1],
                  icos_coords[icos_surface[i][j]][2]);
            v.normalize();
            v.getspherical(phi[j],theta[j],dummy);
// v2.setspherical(phi[j],theta[j],dummy);

// cerr≪"Orig:"≪v≪endl;
// cerr≪"New:"≪v2≪endl;
// cerr≪"Angle:   "≪phi[j]≪","≪theta[j]≪endl;
// cerr≪"Difference:"≪v-v2≪endl≪endl;
        }

        for (j=0;j<8;j++) {
            psi[0]=ipsi[j][0];
            psi[1]=ipsi[j][1];
            child[s]=new Rotationsegment <Data> (phi,theta,psi);
            s++;
        }
    }
    assert(s==ROTATIONSPACE_CHILDREN);
}
```

score.hh

```
// **************************************************
// File: score.hh
// purpose: scoring function scaling
//
// author: Hans-Henrik Staerfeldt
// notes:
```

```
#ifndef SCORE_HH
#define SCORE_HH

#include "gridset.hh"
#include "grid.hh"

void scoresum(const LigandGridSet &L,CGrid &result);
void score(const ReceptorGridSet &R,const LigandGridSet &L,CGrid &result);
void scorecorr(const ReceptorGridSet &Rfft,LigandGridSet &L);

extern double LJfield_scaling;
extern double Desol_scaling;
extern double Elec_scaling;
extern double Shape_scaling;

#endif
```

score.cc

```
// ************************************************
// File: score.cc
// purpose: scaling and combination of correlation functions
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "score.hh"

double LJfield_scaling=0.01; // TODO find good parameter
double Desol_scaling=1.0; // from article
double Elec_scaling=0.006; // from article
double Shape_scaling=0.01; // from article: 0.01

//
// Central scoring function
//
void score(const ReceptorGridSet &R,const LigandGridSet &L,CGrid &result) {
   static CGrid tmpR;
   static CGrid tmpL;
   static CGrid tmpCG;
   int i;

#ifdef DO_Shape
   // by doing this _here_, the _correct_ assignment operator is used!
   tmpL=L.ShapeL;
   tmpR=R.ShapeR;

   // calculate shape complementarity
   tmpCG.corr(tmpR,tmpL);
   for (i=0;i<result.getsizecubed();i++) {
      result(i).re=(tmpCG(i).re-tmpCG(i).im)*Shape_scaling;
      result(i).im=0.0;
   }
#else
   result.clear();
#endif

#ifdef DO_Desol
   tmpR=R.DesolR;
   tmpL=L.DesolL;

   tmpCG.corr(tmpR,tmpL);
   for (i=0;i<result.getsizecubed();i++)
     result(i).re+=tmpCG(i).im*Desol_scaling;
#endif

#ifdef DO_Elec
   tmpR=R.Elecfield;
   tmpL=L.Elecd;

   tmpCG.corr(tmpR,tmpL);
   for (i=0;i<result.getsizecubed();i++)
     result(i).re+=tmpCG(i).re*Elec_scaling;
#endif

#ifdef DO_LJfield
   tmpR=R.LJfield;
   tmpL=L.Density;

   tmpCG.corr(tmpR,tmpL);
   for (i=0;i<result.getsizecubed();i++)
     result(i).re+=tmpCG(i).re*LJfield_scaling;
#endif

}

//
// Scoring that uses precalculated fft of receptor, and
// ruins the ligand in the process (its much faster)
//

void scoresum(const LigandGridSet &L,CGrid &result) {
   int i;
   for (i=0;i<result.getsizecubed();i++) {
      result(i).im=0.0;
      result(i).re=0.0;
#ifdef DO_Shape
      result(i).re=(L.ShapeL(i).re-L.ShapeL(i).im)*Shape_scaling;
#endif
#ifdef DO_Desol
      result(i).re+=L.DesolL(i).im*Desol_scaling;
#endif
#ifdef DO_Elec
      result(i).re+=L.Elecd(i).re*Elec_scaling;
#endif
#ifdef DO_LJfield
      result(i).re+=L.Density(i).re*LJfield_scaling;
#endif
   }
}

void scorecorr(const ReceptorGridSet &Rfft,LigandGridSet &L) {

   L.fft();

#ifdef DO_Shape
   // calculate shape complementarity
   L.ShapeL.multiply(Rfft.ShapeR,L.ShapeL);
#endif

#ifdef DO_Desol
   L.DesolL.multiply(Rfft.DesolR,L.DesolL);
#endif

#ifdef DO_Elec
   L.Elecd.multiply(Rfft.Elecfield,L.Elecd);
#endif

#ifdef DO_LJfield
   L.Density.multiply(Rfft.LJfield,L.Density);
#endif

   L.ifft();

   int i,j;
   int x,y,z;
   double scaledown=1.0/double(L.sizecubed);
   int size=L.size;
   int sizehalf=L.sizehalf;
   Complex c;

   // translation reversal, rescaling in one go.
   for (i=0,x=0;x<size;x++){
      for (y=0;y<size;y++){
         for (z=0;z<size;z++,i++){
            j=L.getidx(size-(x+sizehalf)%size,
                       size-(y+sizehalf)%size,
                       size-(z+sizehalf)%size);
            if (i>j) continue;

#ifdef DO_Shape
            c=L.ShapeL(i);
            L.ShapeL(i)=L.ShapeL(j)*scaledown;
            L.ShapeL(j)=c*scaledown;
#endif

#ifdef DO_Desol
            c=L.DesolL(i);
            L.DesolL(i)=L.DesolL(j)*scaledown;
            L.DesolL(j)=c*scaledown;
#endif

#ifdef DO_Elec
            c=L.Elecd(i);
            L.Elecd(i)=L.Elecd(j)*scaledown;
            L.Elecd(j)=c*scaledown;
#endif

#ifdef DO_LJfield
            c=L.Density(i);
            L.Density(i)=L.Density(j)*scaledown;
            L.Density(j)=c*scaledown;
#endif
         }
      }
   }
}
```

filter.hh

```
// ************************************************
// File: filter.hh
// purpose: filtering of ranked configurational
// results from docking
// author: Hans-Henrik Staerfeldt
// notes:

#include "vector3d.hh"
#include "grid.hh"
#include <vector>
#include <assert.h>

using namespace std;

#define FILTER_RMIN 2.0
#define FILTER_RMAX 8.0
#define FILTER_MAXSIZE 128

typedef struct resultpoint_ {
   double v;
   int idx; // position index
   double ridx; // rotation index
} resultpoint;

int cmpresultpoint(const void *e1, const void *e2);

void filterresults(CGrid &ResultG, int ridx,
                   const int maxresult,
                   vector <resultpoint> &ResultList);

void sortresults(vector <resultpoint> &ResultList);

void getresults(const CGrid &ResultG, int ridx,
                vector <resultpoint> &ResultList);

void limitresults(int maxresults, vector <resultpoint> &ResultList);

double measureresults(const CGrid &ResultG, const Vector3D &correct,
                      const vector <resultpoint> &ResultList );

void showresults(const CGrid &ResultG,
                 const vector <Vector3D> &angles,
                 vector <resultpoint> &ResultList);

void showresults(const CGrid &ResultG,
                 vector <resultpoint> &ResultList);
```

filter.cc

```
// ************************************************
// File:
// purpose: implementation of ranked configurational
// filtering of results from docking
// author: Hans-Henrik Staerfeldt
// notes:

#include "filter.hh"

int cmpresultpoint(const void *e1, const void *e2){
   resultpoint *d1=(resultpoint *)e1;
   resultpoint *d2=(resultpoint *)e2;
   if ( d1->v < d2->v ) return 1;
   if ( d1->v > d2->v ) return -1;
   return 0;
}

void getresults(const CGrid &ResultG, int ridx,
                vector <resultpoint> &ResultList) {
   int i;
   resultpoint r;
   for (i=0;i<ResultG.getsizecubed();i++){
      r.v=ResultG(i).re;
      r.idx=i;
      r.ridx=ridx;
      ResultList.push_back(r);
   }
}

void limitresults(int maxresults, vector <resultpoint> &ResultList) {
   while (ResultList.size()>=maxresults)
     ResultList.pop_back();
}

void filldecent(const int x, const int y, const int z,
                const Vector3D &center, CGrid &grid) {
   static double rmin=FILTER_RMIN;
   static double rmax=FILTER_RMAX;
   double d;

   grid(x,y,z).im=1.0+grid(x,y,z).re; // mark as used in grid
// cerr<<"f"<<x<<"<"<<y<<"<"<<z<<endl;;

   if (x<grid.getsize()-1 && grid(x+1,y,z).im==0.0) {
      d=Vector3D::distance(center,grid.getvector(x+1,y,z));
      if (d<rmin || (d<rmax && grid(x,y,z).re>grid(x+1,y,z).re)) {
         // fill inside rmin regardless, or inside rmax, if on a decent
         filldecent(x+1,y,z,center,grid);
      }
   }

   if (y<grid.getsize()-1 && grid(x,y+1,z).im==0.0) {
      d=Vector3D::distance(center,grid.getvector(x,y+1,z));
      if (d<rmin || (d<rmax && grid(x,y,z).re>grid(x,y+1,z).re)) {
         // fill inside rmin regardless, or inside rmax, if on a decent
         filldecent(x,y+1,z,center,grid);
      }
   }

   if (z<grid.getsize()-1 && grid(x,y,z+1).im==0.0) {
      d=Vector3D::distance(center,grid.getvector(x,y,z+1));
      if (d<rmin || (d<rmax && grid(x,y,z).re>grid(x,y,z+1).re)) {
         // fill inside rmin regardless, or inside rmax, if on a decent
         filldecent(x,y,z+1,center,grid);
      }
   }

   if (x>0 && grid(x-1,y,z).im==0.0) {
      d=Vector3D::distance(center,grid.getvector(x-1,y,z));
      if (d<rmin || (d<rmax && grid(x,y,z).re>grid(x-1,y,z).re)) {
         // fill inside rmin regardless, or inside rmax, if on a decent
         filldecent(x-1,y,z,center,grid);
      }
   }

   if (y>0 && grid(x,y-1,z).im==0.0) {
      d=Vector3D::distance(center,grid.getvector(x,y-1,z));
      if (d<rmin || (d<rmax && grid(x,y,z).re>grid(x,y-1,z).re)) {
         // fill inside rmin regardless, or inside rmax, if on a decent
         filldecent(x,y-1,z,center,grid);
      }
   }

   if (z>0 && grid(x,y,z-1).im==0.0) {
      d=Vector3D::distance(center,grid.getvector(x,y,z-1));
      if (d<rmin || (d<rmax && grid(x,y,z).re>grid(x,y,z-1).re)) {
         // fill inside rmin regardless, or inside rmax, if on a decent
         filldecent(x,y,z-1,center,grid);
      }
   }
}

void filterresults(CGrid &ResultG, int ridx, const int maxresult,
                   vector <resultpoint> &ResultList) {
   static resultpoint *array=NULL;
   static int arraysize=0;

   int i,x,y,z,idx;
   int maxidx;
   int resultcnt;
   int goodcount;
   int size;
   Vector3D maxvector;

   size=ResultG.getsizecubed();
   if (arraysize<size) {
      if (array) delete array;
      array = new resultpoint[size];
      arraysize = size;
   }

   // install in array
   cerr<<"sorting results"<<endl;
   goodcount=0;
   for (i=0;i<size;i++){
      if (ResultG(i).re<0.0) continue;
      array[goodcount].v=ResultG(i).re;
      array[goodcount].idx=i;
      array[goodcount].ridx=ridx;
      goodcount++;
   }

   qsort (array, goodcount, sizeof(resultpoint), cmpresultpoint);

   // only examine the first size entries.
   cerr<<"collecting topscores ("<<goodcount<<")"<<endl;
   resultcnt=0;
   for (maxidx=0;maxidx<goodcount;maxidx++) {
      // find maximum, then fill in points at a
```

```
        // decending grade from that point
        idx=array[maxidx].idx;

        if (ResultG(idx).im!=0.0) continue; //covered already
        maxvector=ResultG.getvector(idx);
        ResultG.getposition(idx,x,y,z);

        // filter results this way
        filldecent(x,y,z,maxvector,ResultG);

        ResultList.push_back(array[maxidx]);
        if (++resultcnt>=maxresult) return;
    }
}

void sortresults(vector <resultpoint> &ResultList) {
    static resultpoint *array=NULL;
    static int arraysize=0;
    int i;
    int size;

    size=ResultList.size();
    if (arraysize<size) {
        if (array) delete array;
        array = new resultpoint[size];
        arraysize = size;
    }

    for (i=0;i<size;i++)
      array[i]=ResultList[i];

    ResultList.clear();

    qsort (array, size, sizeof(resultpoint), cmpresultpoint);

    for (i=0;i<size;i++)
      ResultList.push_back(array[i]);
}

void showresults(const CGrid &ResultG,
                 const vector <Vector3D> &angles,
                 vector <resultpoint> &ResultList) {
    int i;
    int x,y,z;
    Vector3D resultvector;
    Vector3D gridcenter(ResultG.getsize()*ResultG.getscale()/2.0,
                        ResultG.getsize()*ResultG.getscale()/2.0,
                        ResultG.getsize()*ResultG.getscale()/2.0);

    for (i=0;i<ResultList.size();i++){
        ResultG.getposition(ResultList[i].idx,x,y,z);
        resultvector=-ResultG.getvector(x,y,z)+gridcenter;
        cout<<"Result "<<i+1<<" x:  "<<x<<" y:  "<<y<<" z:  "<<z;
        cout<<" "<<resultvector;
        cout<<" phi:  "<<angles[ResultList[i].ridx].x();
        cout<<" theta:  "<<angles[ResultList[i].ridx].y();
        cout<<" psi:"<<angles[ResultList[i].ridx].z()<<endl;
    }
}

double measureresults(const CGrid &ResultG,
                      const Vector3D &correct,
                      const vector <resultpoint> &ResultList ){
    int count;
    int i;
    double mindist=10000;
    Vector3D gridcenter(ResultG.getsize()*ResultG.getscale()/2.0,
                        ResultG.getsize()*ResultG.getscale()/2.0,
                        ResultG.getsize()*ResultG.getscale()/2.0);
    Vector3D pos;

    count=ResultList.size();
    for (i=0;i<count;i++) {
        pos=ResultG.getvector(ResultList[i].idx);
        if ( Vector3D::distance(pos,correct)<mindist ) {
            mindist=Vector3D::distance(pos,correct);
        }
    }
    return mindist;
}

void showresults(const CGrid &ResultG,
                 vector <resultpoint> &ResultList) {
    int i;
    int x,y,z;
    Vector3D resultvector;
    Vector3D gridcenter(ResultG.getsize()*ResultG.getscale()/2.0,
                        ResultG.getsize()*ResultG.getscale()/2.0,
                        ResultG.getsize()*ResultG.getscale()/2.0);

    for (i=0;i<ResultList.size();i++){
        ResultG.getposition(ResultList[i].idx,x,y,z);
        resultvector=-ResultG.getvector(x,y,z)+gridcenter;
        cout<<"Result "<<i+1<<" x:  "<<x<<" y:  "<<y<<" z:  "<<z;
        cout<<" "<<resultvector<<endl;
    }
}
```

stree.hh

```
// *************************************************
// File: stree.hh
// purpose: space partitioning tree
//
// author: Hans-Henrik Staerfeldt
// notes:

#ifndef STREE_HH
#define STREE_HH

#include <set>
#include <vector>
#include "vector3d.hh"

using namespace std;

template <class Data> class StreeNode;

// *************************************************
// template class: StreeElement
// purpose: An element to find in the space partition tree
// notes: contains pointer to related data

template <class Data> class StreeElement {
 public:
    Vector3D pos; /* position */
    Data *d;                                                     /* data */
    char mark;

    StreeElement(const Vector3D &ipos,Data &d);
    ~StreeElement(void);
    void Update(const Vector3D &ipos);
    int intersect(const Vector3D &min, const Vector3D &max);
    Data *getData(void);
#ifdef DEBGUG
    void Render();
#endif
};

/*
    A level of the world tree
*/
#define Stree_RootNode 0
#define Stree_Node 1
#define Stree_Leaf 2

#define Stree_NCHILDREN 8

struct PtrCmp{
    bool operator()(const void* e1, const void* e2) const
    {
        return ((unsigned long)(e1) - (unsigned long) (e2))>0;
    };
};

// *************************************************
// template class: StreeNode
// purpose: node in the spacepartition tree
// notes:

template <class Data> class StreeNode {
 public:
    Vector3D min; /* rectangle covered by this node */
    Vector3D max;
    unsigned long count;                 /* number of elements under this node */
    int type;                                                /* type of node */

    /* no we cannot make them into a union */
    StreeNode <Data> * child[Stree_NCHILDREN];     /* children to the treenode */
    set <StreeElement <Data> *,PtrCmp> leaf;       /* elements to the tree leaf */

    void Split(void);
    void Merge(void);

    void MakeNode(void);
    void MakeLeaf(void);
    int intersect(const Vector3D &ipos, const double r);
    int intersect(const Vector3D &qmin, const Vector3D &qmax, const double r);
    int inside(const Vector3D &qmin, const Vector3D &qmax, const double r);
```

```
    void Query(const Vector3D &ipos,const double r,
               vector <StreeElement <Data> *> &qresult);
    int Querybool(const Vector3D &qmin, const Vector3D &qmax,const double r);
    void Add(StreeElement <Data> *e);
    void print(ostream &s,const int level);

    void Remove(StreeElement <Data> *e);
    void Remove(int i);
    StreeNode(const Vector3D &min, const Vector3D &max,const int type);
    ~StreeNode(void);
};

// ************************************************
// template class: Stree
// purpose: spacepartition tree
// notes:

template <class Data> class Stree {
 private:
    StreeNode <Data> *root;
    Vector3D min; /* rectangle covered by this tree */
    Vector3D max;
    StreeElement <Data> **result;
    int resultsize;

 public:
    void Query(const Vector3D &ipos,const double r,vector <Data *> &qresult);
    int Querybool(const Vector3D &min, const Vector3D &max, const double r);
    void print(ostream &s);
    inline int size(void) const { return root→count; };
    StreeElement <Data> *Add(const Vector3D &ipos,Data &d);
    StreeElement <Data> *Add(StreeElement <Data> *e);

    StreeElement <Data> *Remove(StreeElement <Data> *e);
    StreeElement <Data> *Update(StreeElement <Data> *e,const Vector3D &ipos);

    Stree(const Vector3D &min, const Vector3D &max);
    ~Stree(void);
};

#endif
```

stree.cc

```
// ************************************************
// File: stree.cc
// purpose: space partitioning tree
//
// author: Hans-Henrik Staerfeldt
// notes: fixed maxsize of 16 is approximating O(log N)

#include "stree.hh"
#include "stream.h"
#include <assert.h>

#define TREESPLIT 16
#define TREESPLITRATIO 2

//#define DEBUGQUERY
//#define DEBUGADD
//#define DEBUG

#ifdef DEBUG
char *debug_spacerbuf=" ";
char *SP="";

#define DEBUGUP {SP++;}
#define DEBUGDOWN {SP--;}
#define DEBUGZERO {SP=&(debug_spacerbuf[19]);}
#endif

template <class Data>
int StreeElement <Data>::intersect(const Vector3D &min,const Vector3D &max) {
    /*
        Test if box intersect element
        */
    if (pos[0]>min[0] && pos[1]>min[1] && pos[2]>min[2] &&
          pos[0]<max[0] && pos[1]<max[1] && pos[2]<max[2]) return 1;
    return 0;
}


template <class Data>
StreeElement <Data>::StreeElement(const Vector3D &ipos,Data &id) {
    pos=ipos;
    d=&id;
    mark=0;
}

template <class Data>
StreeElement <Data>::~StreeElement(void){
}

template <class Data>
void StreeElement <Data>::Update(const Vector3D &ipos) {
    pos=ipos;
}

template <class Data>
Data *StreeElement <Data>::getData(void){
    return d;
};

template <class Data>
StreeNode <Data>::StreeNode(const Vector3D &imin,
                            const Vector3D &imax,
                            const int itype) {
    int i;
    min=imin;
    max=imax;
    count=0;
    type=itype;
    for (i=0;i<Stree_NCHILDREN;i++){
        child[i]=NULL;
    }
    if (type==Stree_RootNode) {
#ifdef DEBUG
        cerr≪SP≪"Creating root node"≪endl;
#endif
        Split();
    }
}

template <class Data>
StreeNode <Data>::~StreeNode(void) {
    int i;
    StreeElement <Data> *e;
    set <StreeElement <Data>*,PtrCmp>::iterator j;

#ifdef DEBUG
    DEBUGDOWN;
#endif

    /* Also delete the subnodes */
    for (i=0;i<Stree_NCHILDREN;i++){
        if (child[i]≠NULL) {
#ifdef DEBUG
            cerr≪SP≪"Deleting child "≪i≪endl;
#endif
            delete child[i];
            child[i]=NULL;
        }
    }

    /* also remove elements */
    while (leaf.size()>0) {
#ifdef DEBUG
        cerr≪SP≪"Deleting leaf elements"≪endl;
#endif
        j=leaf.begin();
        e=*j;
        leaf.erase(j);
        delete e;
    }

#ifdef DEBUG
    DEBUGUP;
#endif

}

// determine if this node intersects a query sphere
template <class Data>
int StreeNode <Data>::intersect(const Vector3D &ipos,const double r) {
    int inside=0;

    // this close approximation includes very little more space
    // than needed as it includes the whole corners, where
    // it only needs to include a semisphere/cylinder.
    // but its really fast.
    if (min.x()-r≤ipos.x() && max.x()+r≥ipos.x()) inside++;
    if (min.y()-r≤ipos.y() && max.y()+r≥ipos.y()) inside++;
    if (min.z()-r≤ipos.z() && max.z()+r≥ipos.z()) inside++;

    if (inside==3) return 1;
    return 0;
}

template <class Data>
int StreeNode <Data>::intersect(const Vector3D &qmin,
```

```
                                    const Vector3D &qmax,
                                    const double r) {

    if (qmax.x()+r<min.x()) return 0;
    if (qmax.y()+r<min.y()) return 0;
    if (qmax.z()+r<min.z()) return 0;
    if (qmin.x()-r>max.x()) return 0;
    if (qmin.y()-r>max.y()) return 0;
    if (qmin.z()-r>max.z()) return 0;

    return 1;
}

template <class Data>
int StreeNode <Data>::inside(const Vector3D &qmin,
                             const Vector3D &qmax,
                             const double r) {

    if (qmax.x()+r>max.x() &&
        qmax.y()+r>max.y() &&
        qmax.z()+r>max.z() &&
        qmin.x()-r<min.x() &&
        qmin.y()-r<min.y() &&
        qmin.z()-r<min.z()) return 1;

    return 0;
}

template <class Data>
void StreeNode <Data>::print(ostream &s,const int level) {
    set <StreeElement <Data>*,PtrCmp>::iterator j;
    static char *spacer=" ";
    char *spaceptr;
    int i;

    spaceptr=&spacer[strlen(spacer)-level];

    switch (type) {
     case Stree_Node:
     case Stree_RootNode:
       for (i=0;i<Stree_NCHILDREN;i++){
           if (child[i]==NULL) continue;
           s<<spaceptr<<"Child "<<i<<"("<<child[i]->type<<")";
           s<<min<<".."<<max<<" "<<child[i]->count<<" entries"<<endl;
           child[i]->print(s,level+1);
       }
       break;
     case Stree_Leaf:
       // check for hits
       for (j=leaf.begin();j!=leaf.end();j++)
          cerr<<spaceptr<<"E. at "<<(*j)->pos<<endl;
       break;
    }

}

template <class Data>
void StreeNode <Data>::Query(const Vector3D &ipos,const double r,
                             vector <StreeElement <Data> *> &qresult) {
    int i;
    set <StreeElement <Data>*,PtrCmp>::iterator j;

#ifdef DEBUG
    DEBUGDOWN;
#endif

    if (!intersect(ipos,r)) {
#ifdef DEBUG
       cerr<<SP<<"Query does not cover child "<<endl;
#endif

#ifdef DEBUG
       DEBUGUP;
#endif
       return;
    }

    switch (type) {
     case Stree_Node:
     case Stree_RootNode:
       for (i=0;i<Stree_NCHILDREN;i++){
           if (child[i]==NULL) continue;
#ifdef DEBUG
           cerr<<SP<<"Query traverse child "<<i<<endl;
#endif
           child[i]->Query(ipos,r,qresult);
       }
       break;
     case Stree_Leaf:
       // check for hits
       for (j=leaf.begin();j!=leaf.end();j++){
           if (Vector3D::distance((*j)->pos,ipos)<r) {
#ifdef DEBUG
               cerr<<SP<<"Query found element"<<endl;
#endif
               if ( !(*j)->mark ) {
                   qresult.push_back((*j));
                   (*j)->mark=1;
               }
           }
       }
       break;
    }
#ifdef DEBUG
    DEBUGUP;
#endif

}

template <class Data>
int StreeNode <Data>::Querybool(const Vector3D &qmin,
                                const Vector3D &qmax,
                                const double r) {
    int i;
    set <StreeElement <Data>*,PtrCmp>::iterator j;

    if (!intersect(qmin,qmax,r)) {
       return 0;
    }

#ifdef DEBUG
    DEBUGDOWN;
#endif

    switch (type) {
     case Stree_Node:
     case Stree_RootNode:
       for (i=0;i<Stree_NCHILDREN;i++){
           if (child[i]==NULL) continue;
           if (child[i]->inside(qmin,qmax,r) && child[i]->count) return 1;
           if (child[i]->intersect(qmin,qmax,r)) {
#ifdef DEBUG
               cerr<<SP<<"Querybool traverse child "<<i<<endl;
#endif
               if (child[i]->Querybool(qmin,qmax,r)) return 1;
           }else{
#ifdef DEBUG
               cerr<<SP<<"Querybool skip child "<<i<<endl;
#endif
           }
       }
       break;
     case Stree_Leaf:

#ifdef DEBUG
       cerr<<SP<<"Querybool check leaf "<<i<<endl;
#endif
       // check for hits
       for (j=leaf.begin();j!=leaf.end();j++)
         if ((*j)->intersect(qmin-r,qmax+r)) return 1; // rough estimate
       break;
    }

#ifdef DEBUG
    cerr<<SP<<"Querybool no hits "<<i<<endl;
#endif // check for hits

#ifdef DEBUG
    DEBUGUP;
#endif
    return 0;
}

template <class Data>
void StreeNode <Data>::Split(void) {

    Vector3D c;
    Vector3D cmin[Stree_NCHILDREN];
    Vector3D cmax[Stree_NCHILDREN];
    int scount [Stree_NCHILDREN];
    int dosplit=0;
    int i;
    set <StreeElement <Data>*,PtrCmp>::iterator j;
    int k;
    StreeElement <Data> *e;

#ifdef DEBUG
    DEBUGDOWN;
#endif
```

```
    if (type==Stree_Node) {
#ifdef DEBUG
        DEBUGUP;
#endif
        return;                                         /* impossible, already split */
    }

    if (!(count>TREESPLIT || type==Stree_RootNode)) {
#ifdef DEBUG
        DEBUGUP;
#endif
        return;                                              /* should not be split */
    }
    /*
        Too many nodes in this leaf, break the leaf up
        .. make sure the breakup helps, or arbandon it
        */

#ifdef DEBUG
    cerr≪SP≪"Split node ("≪count≪")"≪endl;
#endif
    /* get center */
    c=(min+max)/2.0;

    /* check if splitting leaf helps */
    if (type==Stree_Leaf || type==Stree_RootNode) {

        cmin[0]=min;
        cmax[0]=c;

        cmin[1].set(c[0],min[1],min[2]);
        cmax[1].set(max[0],c[1],c[2]);

        cmin[2].set(min[0],c[1],min[2]);
        cmax[2].set(c[0],max[1],c[2]);

        cmin[3].set(c[0],c[1],min[2]);
        cmax[3].set(max[0],max[1],c[2]);

        cmin[4].set(min[0],min[1],c[2]);
        cmax[4].set(c[0],c[1],max[2]);

        cmin[5].set(c[0],min[1],c[2]);
        cmax[5].set(max[0],c[1],max[2]);

        cmin[6].set(min[0],c[1],c[2]);
        cmax[6].set(c[0],max[1],max[2]);

        cmin[7]=c;
        cmax[7]=max;
        /* when is it feasable to break up nodes?
            when the difference in subnodes among the children
            is more than TREESPLITRATIO times more in one than
            in another, for any two children
            */

        if (type≠Stree_RootNode) {

            for (i=0;i<Stree_NCHILDREN;i++){
                scount[i]=0;
                for (j=leaf.begin();j≠leaf.end();j++){
                    e=*j;
                    if (e→intersect(cmin[i],cmax[i])){
                        scount[i]++;
                    }
                }
            }

            dosplit=0;
            for (i=0;i<Stree_NCHILDREN;i++){
                for (k=0;k<Stree_NCHILDREN;k++){
                    if (i==k) continue;
                    if ( (scount[i]+1)/(scount[k]+1) ≥ TREESPLITRATIO ) {
                        dosplit=1;
                        break;
                    }
                }
                if (dosplit) break;
            }

            /* we don't split */
#ifdef DEBUG
            cerr≪SP≪"Do split:  "≪dosplit≪endl;
#endif
            if (!dosplit) {
#ifdef DEBUG
                DEBUGUP;
#endif
                return;
            }
        }
    }                                   /* otherwise i assume it is a root node to be split */

    /* split up the node */
    for (i=0;i<Stree_NCHILDREN;i++) {
#ifdef DEBUG
        cerr≪SP≪"Split create child "≪i≪endl;
#endif
        child[i]=new StreeNode(cmin[i],cmax[i],Stree_Leaf);

        for (j=leaf.begin();j≠leaf.end();j++){
            e=*j;
            if (e→intersect(cmin[i],cmax[i])){
#ifdef DEBUG
                cerr≪SP≪"Split add element to child "≪i≪endl;
#endif
                child[i]→Add(e);
            }
        }

        if (type==Stree_Leaf)
          type=Stree_Node;
    }

    leaf.clear();

#ifdef DEBUG
    DEBUGUP;
#endif

}

template <class Data>
void StreeNode<Data>::Merge(void) {
    StreeNode <Data> *n;
    set <StreeElement <Data>*,PtrCmp>::iterator j;

    if (type==Wtree_RootNode) return;                         /* should not be merged */
    if (count>TREEMERGE) return;                              /* should not be merged */

#ifdef DEBUG
    DEBUGDOWN;
    cerr≪SP≪"Merge node"≪endl;
#endif
    leaf.clear(); // need this?
    for (i=0;i<Stree_NCHILDREN;i++) {
        /*
         * since merge is called after removing the element from
         * all subnodes, the children are expected to be leafnodes all.
         */

        if (child[i]→type≠Stree_Leaf) child[i]→Merge();
        for (j=child[i]→leaf.begin();j≠child[i]→leaf.end();j++) {
#ifdef DEBUG
            cerr≪SP≪"Merge child element"≪endl;
#endif
            leaf.insert(*j);
        }
        delete child[i];
        child[i]=0;
    }

    /* now this node is a proper leaf node */
    MakeLeaf();
    if (count≠leaf.size()) {
        static char error[256];
        sprintf(error,"Error merging returned %d/%d\n",>count,leaf.size());
        throw error;
    }
#ifdef DEBUG
    DEBUGUP;
#endif

}

template <class Data> void StreeNode <Data>::MakeNode(void) {
    type=Stree_Node;
}

template <class Data> void StreeNode <Data>::MakeLeaf(void) {
    type=Stree_Leaf;
}

template <class Data>
void StreeNode <Data>::Add(StreeElement <Data> *e) {
    int i;
    count++;

#ifdef DEBUG
    DEBUGDOWN;
```

```
#endif
    switch (type) {
     case Stree_RootNode:
     case Stree_Node:
        for (i=0;i<Stree_NCHILDREN;i++) {
            if (child[i]==NULL) continue;
            if (e→intersect(child[i]→min,child[i]→max)) {
#ifdef DEBUG
                cerr≪SP≪"Add element to child "≪i≪endl;
#endif
                child[i]→Add(e);
            }
        }
        break;
     case Stree_Leaf:
#ifdef DEBUG
        cerr≪SP≪"Add element to leaf "≪endl;
#endif
        leaf.insert(e);
        Split();
        break;
    }
#ifdef DEBUG
    DEBUGUP;
#endif

}

template <class Data>
void StreeNode <Data>::Remove(StreeElement <Data> *e) {
    int i;

#ifdef DEBUG
    DEBUGDOWN;
#endif
    if (!e→intersect(min,max)) {
#ifdef DEBUG
        DEBUGUP;
#endif
        return;
    }

    switch (type) {
     case Stree_Node:
     case Stree_RootNode:
        for (i=0;i<Stree_NCHILDREN;i++){
            if (child[i]==NULL) continue;
            if (e→intersect(child[i]→min,child[i]→max)) {
#ifdef DEBUG
                cerr≪SP≪"Remove element from child "≪i≪endl;
#endif
                child[i]→Remove(e);
            }
        }
        Merge();
        break;
     case Stree_Leaf:
#ifdef DEBUG
        cerr≪SP≪"Remove element from leaf "≪endl;
#endif
        leaf.remove(&e);
        break;
    }
    count--;
#ifdef DEBUG
    DEBUGUP;
#endif
}

template <class Data>
Stree<Data>::Stree(const Vector3D &imin,
                   const Vector3D &imax) {

    min=imin;
    max=imax;
    result=NULL;
    resultsize=0;

#ifdef DEBUG
    DEBUGZERO;
    cerr≪SP≪"Create tree"≪endl;
#endif
    root=new StreeNode<Data>(min,max,Stree_RootNode);
}

template <class Data> Stree<Data>::~Stree(void){
    delete root;
    root=NULL;
}

template <class Data>
void Stree<Data>::Query(const Vector3D &ipos,
                        const double r,
                        vector <Data *> &qresult) {
    int i;
    vector <StreeElement <Data> *> result;

#ifdef DEBUG
    DEBUGZERO;
#endif

    root→Query(ipos,r,result);

    // copy and clear marks
    qresult.clear();
    for (i=0;i<result.size();i++){
        qresult.push_back(result[i]→d);
        result[i]→mark=0;
    }
}

template <class Data>
int Stree<Data>::Querybool(const Vector3D &qmin,
                           const Vector3D &qmax,
                           const double r) {
#ifdef DEBUG
    DEBUGZERO;
    cerr≪"Query range "≪qmin≪".."≪qmax≪endl;
#endif
    return root→Querybool(qmin,qmax,r);
}


template <class Data>
StreeElement <Data> *Stree<Data>::Add(StreeElement <Data> *e){
#ifdef DEBUG
    DEBUGZERO;
    cerr≪"Add element to tree"≪endl;
#endif
    root→Add(e);
    return e;
}

template <class Data>
StreeElement <Data> *Stree<Data>::Add(const Vector3D &ipos,Data &d) {
    StreeElement <Data> *e;
    e=new StreeElement<Data>(ipos,d);
#ifdef DEBUG
    DEBUGZERO;
#endif
    return Add(e);
}

template <class Data>
StreeElement <Data> *Stree<Data>::Remove(StreeElement <Data> *e){
#ifdef DEBUG
    DEBUGZERO;
    cerr≪"Remove element from tree"≪endl;
#endif
    return root→Remove(e);
}

template <class Data>
void Stree <Data>::print(ostream &s) {
    s≪"Stree"≪endl;
    root→print(s,1);
}

template <class Data>
StreeElement <Data> *Stree<Data>::Update(StreeElement <Data> *e,
                                         const Vector3D &ipos){
    StreeElement <Data> *r;

#ifdef DEBUG
    DEBUGZERO;
    cerr≪"Update element in tree"≪endl;
#endif
    r=Remove(e);
    if (r≠NULL) {
        r→Update(ipos);
        Add(r);
    }
    return r;
}
```

structure.hh

```
// **************************************************
// File: structure.hh
```

```
// purpose: protein structure containment and manipulation
// author: Hans-Henrik Staerfeldt
// notes:

#ifndef __STRUCTURE_H__
#define __STRUCTURE_H__

#include <pdb++.h>
#include <vector>
#include <iostream.h>
#include "vector3d.hh"
#include "grid.hh"
#include "stree.hh"
#include "gridset.hh"

using namespace std;

#include "atoms.hh"

#include "residues.hh"

#define ALTERNATIVEALGORITHMS
// maximal number of actual atoms in a residue
#define RESIDUE_MAXATOM 27

// ************************************************
// struct: ResidueInfo
// purpose: information about residues
// notes:

typedef struct ResidueInfo_ {
   char *name;
   char key[4];
   int atomcount;
   ResidueAtomtype atomtype[RESIDUE_MAXATOM+1];
   double charge[RESIDUE_MAXATOM+1];
   double vdwradius[RESIDUE_MAXATOM+1];
   double vdwwell[RESIDUE_MAXATOM+1];
   int atomnum[RATOM_MAXTYPE+1];
} ResidueInfo;

extern ResidueInfo ResidueInfoData[];
extern ResidueInfo ResidueInfoDatact[];
extern ResidueInfo ResidueInfoDatant[];

// ************************************************
// struct: AtomInfo
// purpose: informatin about atom types
// notes: only contains their key

typedef struct AtomInfo_ {
   char key[6];
} AtomInfo;

extern AtomInfo AtomInfoData[];
extern char *PlacementInfoData[];

// ************************************************
// class: Atom
// purpose: data and methods for an atom in a protein structure
// notes:

class Atom {
   friend class Residue;
   friend class Protein;
   friend istream &operator>>(istream &s, Protein &p);
   friend ostream &operator<<(ostream &s, Protein &p);

 private:
   char altLoc;
   double occupancy;
   double tempFactor;
   int ftnoteNum;
   int serialNum;

   void write(ostream &s,const char *name,const char chainId,
              const int seqNum,const char insertCode);

 public:
   double charge;
   double vdwradius;
   double vdwwell;
   double surface;
   Vector3D pos;
   ResidueAtomtype type;
   Residuetype residuetype;

   Atom();
   ~Atom();
   void clear();
   inline char getaltLoc() { return altLoc;};
   inline double getoccupancy() { return occupancy;};
   inline double gettempFactor() { return tempFactor;};
   inline int getftnoteNum() { return ftnoteNum;};
   inline int getserialNum() { return serialNum;};

   int settype(const char *typestring);
   void setvalues(const int serialNum,
                  const char altLoc,
                  const double occupancy,
                  const double tempFactor,
                  const int ftnoteNum);
   void transform(const Matrix4D &matrix);
   void translate(const Vector3D &offset);
   void getcog(Vector3D &sum, int &count);
   void getradius(const Vector3D &cog, double &r1, double &r2);
   void getbbox(Vector3D &p1,Vector3D &p2);
   void render(ReceptorGridSet &mygrid);
   void render(LigandGridSet &mygrid);
};


enum ResiduePlacementtype {
   RESIDUE_CTERMINUS=0,
   RESIDUE_CENTER,
   RESIDUE_NTERMINUS
};

// ************************************************
// class: Residue
// purpose: data and methods for a residue in a protein
// notes:

class Residue {
   friend istream &operator>>(istream &s, Protein &p);
   friend ostream &operator<<(ostream &s, Protein &p);
   friend class Protein;
 private:
   char chainId;
   int seqNum;
   char insertCode;

   void write(ostream &s);
   ResiduePlacementtype placement;

 public:
   Atom atom[RESIDUE_MAXATOM];
   Residuetype type;
   double surface;


   Residue(void);
   ~Residue(void);
   void clear();
   inline int size() {
      switch(placement){
       case RESIDUE_NTERMINUS:
         return ResidueInfoDatant[type].atomcount;
       case RESIDUE_CENTER:
         return ResidueInfoData[type].atomcount;
       case RESIDUE_CTERMINUS:
         return ResidueInfoDatact[type].atomcount;
      }
      return 0; // should never happend
   }
   inline ResiduePlacementtype getplacement() { return placement; };
   inline char getchainId() { return chainId;};
   inline int getseqNum() { return seqNum;};
   inline char getinsertCode() { return insertCode;};

   Atom &operator[](const ResidueAtomtype atype);
   inline Atom &operator[](const int atype) {
      return Residue::operator[]( ResidueAtomtype(atype));
   }
   void collectatoms(Stree <Atom> &atomtree);
   void collectatoms(vector <Atom *> &atoms);
   void addAtom(const Atom &a);
   void addAtom(const ResidueAtomtype rtype,
                const double bond, const double theta, const double phi,
                const ResidueAtomtype c1, const ResidueAtomtype c2,
                const ResidueAtomtype c3);
   void setvalues(const char chainId,
                  const int seqNum,
                  const char insertCode);
   int settype(const char *typestring,ResiduePlacementtype iplacement);
   void buildhydrogen(const double bb_phi, const double bb_psi);
   void buildchi(const double bb_phi,
                 const double bb_psi,
                 const float dchi[4]);
   void transform(const Matrix4D &matrix);
```

```
    void translate(const Vector3D &offset);
    void getcog(Vector3D &sum, int &count);
    void getradius(const Vector3D &cog, double &r1, double &r2);

    void readatom();
    void getbbox(Vector3D &p1, Vector3D &p2);
    void render(ReceptorGridSet &mygrid);
    void render(LigandGridSet &mygrid);
};

// ************************************************
// class: Protein
// purpose: structure and methods for an entire protein
// notes:

class Protein {
friend istream &operator>>(istream &s, Protein &p);
friend ostream &operator<<(ostream &s, Protein &p);
  private:
    vector <PDB> header;
    double surfaceball;
    ResiduePlacementtype single_residue_placement;
#ifdef ALTERNATIVEALGORITHMS
    void rendertreerecurse(Stree <Atom> &atomtree,
                           ReceptorGridSet &mygrid,
                           const int min[3],
                           const int max[3]);
    void rendertreeleaf(Stree <Atom> &atomtree,
                        ReceptorGridSet &mygrid,
                        const int min[3],
                        const int max[3]);
    void rendertreerecurse(Stree <Atom> &atomtree,
                           LigandGridSet &mygrid,
                           const int min[3],
                           const int max[3]);
    void rendertreeleaf(Stree <Atom> &atomtree,
                        LigandGridSet &mygrid,
                        const int min[3],
                        const int max[3]);
#endif
    void renderdelta(ReceptorGridSet &grid);
    void renderdelta(LigandGridSet &grid);
  public:
    vector <Residue> residue;

    Protein(void);
    ~Protein(void);
    void clear();
    inline void setsingleresidueplacement(const ResiduePlacementtype placement)
    { single_residue_placement=placement; };
    inline ResiduePlacementtype getsingleresidueplacement(void)
    { return single_residue_placement; };

    void collectatoms(Stree <Atom> &atomtree);
    void collectatoms(vector <Atom *> &atoms);
    void findsurface(double radius);

    void addResidue(const Residue &r);
    void transform(const Matrix4D &matrix);
    void translate(const Vector3D &offset);

    void render(ReceptorGridSet &mygrid);
    void render(LigandGridSet &mygrid);
#ifdef ALTERNATIVEALGORITHMS
    void rendertree(ReceptorGridSet &mygrid);
    void rendertreefast(ReceptorGridSet &mygrid);
    void rendertree(LigandGridSet &mygrid);
    void rendertreefast(LigandGridSet &mygrid);
#endif
    void save(char *filename);
    void load(char *filename);
    void rechain(char newchainId);
    void rechain(char oldchainId, char newchainId);

    Protein getchain(char chainId);
    Protein getchains(char *chainIds);
    void merge(const Protein &p);
    void buildhydrogen(void);
    void getbbox(Vector3D &p1, Vector3D &p2);
    double getradius(const Vector3D &center);
    double getdiameter(const Vector3D &center);
    Vector3D getcog();

};

istream &operator>>(istream &s,const Protein &p);
ostream &operator<<(ostream &s,const Protein &p);

#endif
```

structure.cc

```
// ************************************************
// File: structure.hh
// purpose: protein structure containment and manipulation
// author: Hans-Henrik Staerfeldt
// notes:

#include <pdb++.h>
#include <strings.h>
#include <math.h>
#include <vector.h>
#include "field.hh"
#include "grid.hh"
#include "structure.hh"
#include "vector3d.hh"

//#define SHOWCHARGE
//#define SHOWSURFACE

using namespace std;
// Autogenerated residue information
//
#include "residues.cc"
#include "residuesnt.cc"
#include "residuesct.cc"

#include "atoms.cc"

char *PlacementInfoData[3]={
    "C-terminus",
    "In chain",
    "N-Terminus"
};

static int ProteinAtomCount;

//
// Protein level implementation
//

Protein::Protein(void) {
    clear();
    single_residue_placement=RESIDUE_CENTER;
    surfaceball=-1;
}

Protein::~Protein(void){
}

void Protein::clear(){
    residue.clear();
    header.clear();
    surfaceball=-1.0;
}

void Protein::save(char *filename) {
    ofstream ofs;
    ofs.open(filename);
    if (ofs.bad()) throw "Could not save structure";
    ofs<<*this;
    ofs.close();
}

void Protein::load(char *filename) {
    ifstream ifs;
    ifs.open(filename);
    if (ifs.bad()) throw "Could not load structure";
    ifs>>*this;
    ifs.close();
}

void Protein::collectatoms(Stree <Atom> &atomtree) {
    int i;
    for (i=0;i<residue.size();i++) {
        residue[i].collectatoms(atomtree);
    }
}

void Protein::collectatoms(vector <Atom *> &atoms) {
    int i;
    for (i=0;i<residue.size();i++) {
        residue[i].collectatoms(atoms);
    }
}

void Protein::transform(const Matrix4D &matrix) {
    int i;
    for (i=0;i<residue.size();i++) {
        residue[i].transform(matrix);
```

```
    }
}

void Protein::translate(const Vector3D &offset) {
    int i;
    for (i=0;i<residue.size();i++) {
        residue[i].translate(offset);
    }
}

Vector3D Protein::getcog() {
    Vector3D r=Vector3D();
    int count=0;
    int i;
    for (i=0;i<residue.size();i++) {
        residue[i].getcog(r,count);
    }
    r=r/count;
    return r;
}

// generate a circle circumcribing the protein
double Protein::getdiameter(const Vector3D &cog){
    int i;
    double r1,r2;
    for (i=0;i<residue.size();i++) {
        residue[i].getradius(cog,r1,r2);
    }
    return r1+r2/2;
}

double Protein::getradius(const Vector3D &cog){
    int i;
    double r1,r2;
    for (i=0;i<residue.size();i++) {
        residue[i].getradius(cog,r1,r2);
    }
    if (r1>r2) return r1;
    return r2;
}

void Protein::getbbox(Vector3D &p1,Vector3D &p2) {
    int i;
    p1.clear();
    p2.clear();
    for (i=0;i<residue.size();i++) {
        residue[i].getbbox(p1,p2);
    }
}

// **************************************************
// method: Protein::rechain
// purpose: change chaid of protein chain
// notes:

void Protein::rechain(char oldchainId, char newchainId) {
    int i;

    for (i=0;i<header.size();i++) {
        switch(header[i].type()) {
         case PDB::SEQRES:
            if (header[i].seqres.chainId==oldchainId)
               header[i].seqres.chainId=newchainId;
            break;
         case PDB::SHEET:
            if (header[i].sheet.residues[0].chainId==oldchainId)
               header[i].sheet.residues[0].chainId=newchainId;
            if (header[i].sheet.residues[1].chainId==oldchainId)
               header[i].sheet.residues[1].chainId=newchainId;
            break;
         case PDB::TURN:
            if (header[i].turn.residues[0].chainId==oldchainId)
               header[i].turn.residues[0].chainId=newchainId;
            if (header[i].turn.residues[1].chainId==oldchainId)
               header[i].turn.residues[1].chainId=newchainId;
            break;
         case PDB::HELIX:
            if (header[i].helix.residues[0].chainId==oldchainId)
               header[i].helix.residues[0].chainId=newchainId;
            if (header[i].helix.residues[1].chainId==oldchainId)
               header[i].helix.residues[1].chainId=newchainId;
            break;
         case PDB::SSBOND:
            if (header[i].ssbond.residues[0].chainId==oldchainId)
               header[i].ssbond.residues[0].chainId=newchainId;
            if (header[i].ssbond.residues[1].chainId==oldchainId)
               header[i].ssbond.residues[1].chainId=newchainId;
            break;
        }
    }
    for (i=0;i<residue.size();i++) {
        if (residue[i].chainId==oldchainId)
           residue[i].chainId=newchainId;
    }
}

void Protein::rechain(char newchainId) {
    int resnum=0;
    int lastseqNum=-999;
    int i;
    // changes chainid and numbering of protein
    header.clear(); // cant keep these
    for (i=0;i<residue.size();i++) {
        residue[i].chainId=newchainId;
        if (lastseqNum≠residue[i].seqNum)
           resnum++;
        lastseqNum=residue[i].seqNum;
        residue[i].seqNum=resnum;
    }
}


void Protein::addResidue(const Residue &r) {
    residue.push_back(r);
}


Protein Protein::getchain(char chainId) {
    Protein r;
    int i;

    for (i=0;i<header.size();i++) {
        switch(header[i].type()) {
         case PDB::SEQRES:
            if (header[i].seqres.chainId==chainId)
               r.header.push_back(header[i]);
            break;
         case PDB::SHEET:
            if (header[i].sheet.residues[0].chainId==chainId ||
                  header[i].sheet.residues[1].chainId==chainId)
               r.header.push_back(header[i]);
            break;
         case PDB::TURN:
            if (header[i].turn.residues[0].chainId==chainId ||
                  header[i].turn.residues[1].chainId==chainId)
               r.header.push_back(header[i]);
            break;
         case PDB::HELIX:
            if (header[i].helix.residues[0].chainId==chainId ||
                  header[i].helix.residues[1].chainId==chainId)
               r.header.push_back(header[i]);
            break;
         case PDB::SSBOND:
            if (header[i].ssbond.residues[0].chainId==chainId ||
                  header[i].ssbond.residues[1].chainId==chainId)
               r.header.push_back(header[i]);
            break;
         default:
            r.header.push_back(header[i]);
        }
    }

    r.single_residue_placement=single_residue_placement;

    for (i=0;i<residue.size();i++)
       if (residue[i].getchainId()==chainId)
          r.addResidue(residue[i]);

    return r;
}

Protein Protein::getchains(char *chainIds) {
    Protein r;
    int i;

    for (i=0;i<header.size();i++) {
        switch(header[i].type()) {
         case PDB::SEQRES:
            if (strchr(chainIds,header[i].seqres.chainId))
               r.header.push_back(header[i]);
            break;
         case PDB::SHEET:
            if (strchr(chainIds,header[i].sheet.residues[0].chainId) ||
                  strchr(chainIds,header[i].sheet.residues[1].chainId))
               r.header.push_back(header[i]);
            break;
         case PDB::TURN:
            if (strchr(chainIds,header[i].turn.residues[0].chainId) ||
                  strchr(chainIds,header[i].turn.residues[1].chainId))
```

```
            r.header.push_back(header[i]);
          break;
        case PDB::HELIX:
          if (strchr(chainIds,header[i].helix.residues[0].chainId) ||
              strchr(chainIds,header[i].helix.residues[1].chainId))
            r.header.push_back(header[i]);
          break;
        case PDB::SSBOND:
          if (strchr(chainIds,header[i].ssbond.residues[0].chainId) ||
              strchr(chainIds,header[i].ssbond.residues[1].chainId))
            r.header.push_back(header[i]);
          break;
        default:
          r.header.push_back(header[i]);
      }
   }

   r.single_residue_placement=single_residue_placement;

   for (i=0;i<residue.size();i++)
     if (strchr(chainIds,residue[i].getchainId()))
       r.addResidue(residue[i]);

   return r;
}


void Protein::merge(const Protein &p) {
   int i;

   for (i=0;i<p.residue.size();i++)
     addResidue(p.residue[i]);

   for (i=0;i<p.header.size();i++)
     header.push_back(p.header[i]);
}

//
// Residue level implementation
//

Residue::Residue(void) {
   clear();
   placement=RESIDUE_CENTER;
}

Residue::~Residue(void) {
}

Atom &Residue::operator[](const ResidueAtomtype atype) {
   int atomnum;
   static char error[256];
   ResidueInfo *ResidueInfoDataPtr;

   switch(placement){
    case RESIDUE_NTERMINUS:
      ResidueInfoDataPtr=ResidueInfoDatant;
      break;
    case RESIDUE_CENTER:
      ResidueInfoDataPtr=ResidueInfoData;
      break;
    case RESIDUE_CTERMINUS:
      ResidueInfoDataPtr=ResidueInfoDatact;
      break;
   }
   atomnum=ResidueInfoDataPtr[type].atomnum[atype];
   if (atomnum==-1) {
      sprintf(error,"Illegal atom type [%s] for residue %s(%d) (%s).",
              AtomInfoData[atype].key,ResidueInfoDataPtr[type].key,
              seqNum,PlacementInfoData[placement]);
      throw error;
   }

   return this->atom[atomnum];
}

void Residue::clear() {
   int i;
   for (i=0;i<RESIDUE_MAXATOM;i++)
     atom[i].clear();
   type=RESIDUE_MAX;
   placement=RESIDUE_CENTER;
}

void Residue::collectatoms(Stree <Atom> &atomtree) {
   int i;
   // collect atoms in tree
   for (i=0;i<size();i++) {
      if (atom[i].type==RATOM_MAX) continue;
      atomtree.Add(atom[i].pos,atom[i]);
   }
}

void Residue::collectatoms(vector <Atom *> &atoms) {
   int i;
   // collect atoms in set
   for (i=0;i<size();i++) {
      if (atom[i].type==RATOM_MAX) continue;
      atoms.push_back(&atom[i]);
   }
}

void Residue::transform(const Matrix4D &matrix) {
   int i;
   for (i=0;i<size();i++) {
      if (atom[i].type==RATOM_MAX) continue; // skip if not there
      atom[i].transform(matrix);
   }
}

void Residue::translate(const Vector3D &offset) {
   int i;
   for (i=0;i<size();i++) {
      if (atom[i].type==RATOM_MAX) continue; // skip if not there
      atom[i].translate(offset);
   }
}

void Residue::getcog(Vector3D &sum, int &count) {
   int i;
   for (i=0;i<size();i++) {
      if (atom[i].type==RATOM_MAX) continue; // skip if not there
      atom[i].getcog(sum,count);
   }
}

void Residue::getradius(const Vector3D &cog, double &r1, double &r2){
   int i;
   for (i=0;i<size();i++) {
      if (atom[i].type==RATOM_MAX) continue; // skip if not there
      atom[i].getradius(cog,r1,r2);
   }
}

void Residue::getbbox(Vector3D &p1, Vector3D &p2) {
   int i;
   for (i=0;i<size();i++) {
      if (atom[i].type==RATOM_MAX) continue; // skip if not there
      atom[i].getbbox(p1,p2);
   }
}

void Residue::write( ostream &s ) {
   int i;
   int atomcnt;
   const char *key;

   ResidueInfo *ResidueInfoDataPtr;

   switch(placement){
    case RESIDUE_NTERMINUS:
      ResidueInfoDataPtr=ResidueInfoDatant;
      break;
    case RESIDUE_CENTER:
      ResidueInfoDataPtr=ResidueInfoData;
      break;
    case RESIDUE_CTERMINUS:
      ResidueInfoDataPtr=ResidueInfoDatact;
      break;
   }

   // rasmol does not like the residue extension codes...
   atomcnt=ResidueInfoDataPtr[type].atomcount;
   key=ResidueInfoDataPtr[type].key;
   for (i=0;i<atomcnt;i++) {
      if (atom[i].type==RATOM_MAX) continue; // skip if not there
      switch (type) {
       case RESIDUE_ASH:
         atom[i].write(s,ResidueInfoDataPtr[RESIDUE_ASN].key,
                       chainId,seqNum,insertCode);
         break;
       case RESIDUE_CYM:
       case RESIDUE_CYX:
         atom[i].write(s,ResidueInfoDataPtr[RESIDUE_CYS].key,
                       chainId,seqNum,insertCode);
         break;
       case RESIDUE_GLH:
         atom[i].write(s,ResidueInfoDataPtr[RESIDUE_GLN].key,
                       chainId,seqNum,insertCode);
         break;
```

```
      case RESIDUE_HID:
      case RESIDUE_HIE:
      case RESIDUE_HIP:
        atom[i].write(s,ResidueInfoDataPtr[RESIDUE_HIS].key,
                      chainId,seqNum,insertCode);
        break;
      case RESIDUE_LYN:
        atom[i].write(s,ResidueInfoDataPtr[RESIDUE_LYS].key,
                      chainId,seqNum,insertCode);
        break;
      default:
        atom[i].write(s,key,chainId,seqNum,insertCode);
      }
  }
}

void Residue::addAtom(const Atom &a) {
  int atomnum;
  static char error[256];
  ResidueInfo *ResidueInfoDataPtr;

  switch(placement) {
   case RESIDUE_CTERMINUS:
     ResidueInfoDataPtr=ResidueInfoDatact;
     break;
   case RESIDUE_CENTER:
     ResidueInfoDataPtr=ResidueInfoData;
     break;
   case RESIDUE_NTERMINUS:
     ResidueInfoDataPtr=ResidueInfoDatant;
     break;
  }

  atomnum=ResidueInfoDataPtr[type].atomnum[a.type];
  if (atomnum==-1) {
     sprintf(error,"Illegal atom type %s for residue %s(%d) (%s)",
             AtomInfoData[a.type].key,ResidueInfoDataPtr[type].key,
             seqNum,PlacementInfoData[placement]);
     throw error;
  }
  // associate correct data when assigned to residue
  atom[atomnum]=a;
  atom[atomnum].residuetype=type;
  atom[atomnum].vdwradius=ResidueInfoDataPtr[type].vdwradius[atomnum];
  atom[atomnum].vdwwell=ResidueInfoDataPtr[type].vdwwell[atomnum];
  atom[atomnum].charge=ResidueInfoDataPtr[type].charge[atomnum];
}

int Residue::settype(const char *typestring,
                     ResiduePlacementtype iplacement) {
  int t;
  static char error[255];
  ResidueInfo *ResidueInfoDataPtr;

  switch (iplacement) {
   case RESIDUE_CTERMINUS:
     ResidueInfoDataPtr=ResidueInfoDatact;
     break;
   case RESIDUE_CENTER:
     ResidueInfoDataPtr=ResidueInfoData;
     break;
   case RESIDUE_NTERMINUS:
     ResidueInfoDataPtr=ResidueInfoDatant;
     break;
  }

  for (t=RESIDUE_START;t<RESIDUE_MAX;t++) {
     if (strcmp(ResidueInfoDataPtr[t].key,typestring)==0) {
        clear();
        type=(Residuetype) t;
        placement=iplacement;
        if (ResidueInfoDataPtr[t].atomcount==0) {
           sprintf(error,"Illegal residue type:%s (%s)",
                   typestring,PlacementInfoData[placement]);
           throw error;
        }
        return 0; // ok
     }
  }

  sprintf(error,"Illegal residue type:%s (%s)",
          typestring,PlacementInfoData[placement]);
  throw error;
}

void Residue::setvalues(const char vchainId,
                        const int vseqNum,
                        const char vinsertCode) {
  chainId=vchainId;
  seqNum=vseqNum;
  insertCode=vinsertCode;
}

//
// Atom level implementation
//
Atom::Atom() {
  clear();
}

Atom::~Atom() {
}

void Atom::clear() {
  type=RATOM_MAX;
  pos.set(0.0,0.0,0.0);
  altLoc=' ';
  occupancy=0.0;
  tempFactor=0.0;
  ftnoteNum=0;
  serialNum=0;
  charge=0.0;
  vdwradius=0.0;
  vdwwell=0.0;
  surface=-1;
  residuetype=RESIDUE_MAX;
}

void Atom::transform(const Matrix4D &matrix) {
  pos=pos*matrix;
}

void Atom::translate(const Vector3D &offset) {
  pos=pos+offset;
}

void Atom::getcog(Vector3D &sum, int &count) {
  count++;
  sum=sum+pos;
}

void Atom::getradius(const Vector3D &cog, double &r1, double &r2){
  double r;
  r=Vector3D::distance(pos,cog);
  if (r>r1) {
     r2=r1;
     r1=r;
  }else if (r>r2){
     r2=r;
  }
}

void Atom::getbbox(Vector3D &p1,Vector3D &p2) {
  if (p1[0]==p2[0] && p1[0]==0.0 && p1[1]==p2[1] &&
      p1[1]==0.0 && p1[2]==p2[2] && p1[2]==0.0) {
     // uninitialized
     p1=pos;
     p2=pos;
  }else{
     if (p1[0]>pos[0]) p1[0]=pos[0];
     if (p1[1]>pos[1]) p1[1]=pos[1];
     if (p1[2]>pos[2]) p1[2]=pos[2];

     if (p2[0]<pos[0]) p2[0]=pos[0];
     if (p2[1]<pos[1]) p2[1]=pos[1];
     if (p2[2]<pos[2]) p2[2]=pos[2];
  }
}

int Atom::settype(const char *typestring) {
  int i;
  char newkey[8];
  static char error[256];

  if (typestring[0]==' ') { // ' CA' → 'CA'
     strcpy(newkey,typestring+1);
  }else{ // '2HH1' → 'HH12'
     newkey[0]=typestring[1];
     newkey[1]=typestring[2];
     newkey[2]=typestring[3];
     newkey[3]=typestring[0];
     newkey[4]='
0';
  }

  if (!*typestring) {
     type=(RATOM_MAX);
     return 0;
  }
```

```
    for (i=0;i<RATOM_MAXTYPE;i++) {
        if (strcmp(AtomInfoData[i].key,newkey)==0) {
            type=(ResidueAtomtype)i;
            return 0; // ok
        }
    }
// cerr ≪ "could not find '"≪newkey≪"'"≪endl;
    sprintf(error,"Unknown atom type %s",newkey);
    throw error;
}

void Atom::setvalues(const int vserialNum,
                     const char valtLoc,
                     const double voccupancy,
                     const double vtempFactor,
                     const int vftnoteNum){

    serialNum=vserialNum;
    altLoc=valtLoc;
    occupancy=voccupancy;
    tempFactor=vtempFactor;
    ftnoteNum=vftnoteNum;
}

void Atom::write(ostream &s,
                 const char *name,
                 const char chainId,
                 const int seqNum,
                 const char insertCode){

    PDB rec;
    char key[6];

    rec.type(PDB::ATOM);
    // rework the atom counters for pdb output
    rec.atom.serialNum=ProteinAtomCount++;

    if (strlen(AtomInfoData[type].key)==4) {
        key[0]=AtomInfoData[type].key[3];
        key[1]=AtomInfoData[type].key[0];
        key[2]=AtomInfoData[type].key[1];
        key[3]=AtomInfoData[type].key[2];
    }else{
        key[0]=' ';
        key[1]=AtomInfoData[type].key[0];
        key[2]=AtomInfoData[type].key[1];
        key[3]=AtomInfoData[type].key[2];
    }
    key[4]='
0';
    strcpy(rec.atom.name,key);
    rec.atom.altLoc=altLoc;
    rec.atom.xyz[0]=pos[0];
    rec.atom.xyz[1]=pos[1];
    rec.atom.xyz[2]=pos[2];

    rec.atom.occupancy=occupancy;
    rec.atom.ftnoteNum=ftnoteNum;

    rec.atom.tempFactor=tempFactor;
#ifdef SHOWCHARGE
    rec.atom.tempFactor=charge;
#endif
#ifdef SHOWSURFACE
    rec.atom.tempFactor=surface;
#endif
    strcpy(rec.atom.residue.name,name);
    rec.atom.residue.chainId=chainId;
    rec.atom.residue.seqNum=seqNum;
    rec.atom.residue.insertCode=insertCode;

    s≪rec≪endl;
}


//
// Protein level input/output from/to PDB entry stream
//

ostream &operator≪(ostream &s, Protein &p) {
    int i;
    // first render original headers;
    ProteinAtomCount=1;
    for (i=0;i<p.header.size();i++){
        s≪p.header[i]≪endl;
    }
    for (i=0;i<p.residue.size();i++) {
        p.residue[i].write(s);
    }
    s≪"END"≪endl;
    return s;
}

// read a protein:
istream &operator≫(istream& s, Protein &p) {
    int last_seqNum=-999;
    char last_chainId='$';
    char last_atomname[8]="XXX";
    char residue_chainId;
    char residue_name[8]="XXX";
    int residue_seqNum;
    char residue_insertCode;
    int i;
    vector <PDB> ssbonds;

    Vector3D point;
    PDB rec;
    Atom a[64];
    int atomcnt=0;
    ResiduePlacementtype placement;
    Residue r;

    p.residue.clear();
    placement=RESIDUE_NTERMINUS;

    while (s ≫ rec) {

        switch (rec.type()) {
         case PDB::SSBOND:
            // example;
            // SSBOND 1 CYS E 1 CYS E 122
            ssbonds.push_back(rec);
            // no break;
         case PDB::SCALE:
         case PDB::SEQRES:
         case PDB::TURN:
         case PDB::SHEET:
         case PDB::HELIX:
         case PDB::SOURCE:
         case PDB::HEADER:
         case PDB::MTRIX:
         case PDB::ORIGX:
         case PDB::REVDAT:
         case PDB::AUTHOR:
         case PDB::COMPND:
         case PDB::CRYST1:
            p.header.push_back(rec);
            break;
         case PDB::ATOM:
            // skip alternate positions (take only first)
            if (atomcnt>1 && !strcmp(rec.atom.name,last_atomname)) {
                cerr≪"Warning; skipped altLoc atom ";
                cerr≪rec.atom.serialNum≪" residue ";
                cerr≪rec.atom.residue.seqNum≪endl;
                continue;
            }

            if (!strcmp(rec.atom.residue.name,"ACE")) {
                cerr≪"Warning; skipped ACE"≪endl;
                continue;
            }
            if (rec.atom.residue.insertCode≠' ') continue;

            if (rec.atom.residue.seqNum≠last_seqNum ||
                 rec.atom.residue.chainId≠last_chainId) {

                // if we see an atom in the new chain, set this to the last
                if (last_chainId≠rec.atom.residue.chainId)
                    placement=RESIDUE_CTERMINUS;

                // if we see a new residue, add the array of atoms.
                if (last_seqNum≠-999) {

                    // set placement/type for new residue.
                    if (!strcmp("CYS",residue_name)) {

                        // is there an s-s bond?
                        for (i=0;i<ssbonds.size();i++) {
                            if ((ssbonds[i].ssbond.residues[0].seqNum ==
                                   residue_seqNum) &&
                                  (ssbonds[i].ssbond.residues[0].chainId ==
                                   residue_chainId) &&
                                  (ssbonds[i].ssbond.residues[0].insertCode ==
                                   residue_insertCode)) {
                                strcpy(residue_name,"CYX");
                                break;
                            }
                            if ((ssbonds[i].ssbond.residues[1].seqNum ==
                                   residue_seqNum) &&
                                  (ssbonds[i].ssbond.residues[1].chainId ==
```

```
                                residue_chainId) &&
                               (ssbonds[i].ssbond.residues[1].insertCode ==
                                residue_insertCode)) {
                              strcpy(residue_name,"CYX");
                              break;
                          }
                      }
                  }

                  r.settype(residue_name,placement);
                  r.setvalues(residue_chainId,
                              residue_seqNum,
                              residue_insertCode);

                  // collect residue from array of atoms
                  for (i=0;i<atomcnt;i++){
                      try {
                          r.addAtom(a[i]);
                      }
                      catch (char *error) {
                          cerr<<"Warning; "<<error<<endl;
                      }

                      a[i].clear();
                  }
                  p.addResidue(r); // add collected residue;
                  r.clear();
              }

              last_seqNum=rec.atom.residue.seqNum;
              last_chainId=rec.atom.residue.chainId;

              // the next is middle
              if (placement==RESIDUE_NTERMINUS)
                  placement=RESIDUE_CENTER;
              // the next is first
              if (placement==RESIDUE_CTERMINUS)
                  placement=RESIDUE_NTERMINUS;

              atomcnt=0;

              strcpy(residue_name,rec.atom.residue.name);
              residue_chainId=rec.atom.residue.chainId;
              residue_seqNum=rec.atom.residue.seqNum;
              residue_insertCode=rec.atom.residue.insertCode;
          }

          strcpy(last_atomname,rec.atom.name);
          try {
              a[atomcnt].settype(rec.atom.name);
              a[atomcnt].pos.set(rec.atom.xyz[0],
                                 rec.atom.xyz[1],
                                 rec.atom.xyz[2]);

              a[atomcnt].setvalues(rec.atom.serialNum,
                                   rec.atom.altLoc,
                                   rec.atom.occupancy,
                                   rec.atom.tempFactor,
                                   rec.atom.ftnoteNum);
              atomcnt++;
          }
          catch (char *error) {
              cerr<<"Warning; "<<error<<endl;
          }

          break;
      }
  }

  if (last_seqNum!=-999) {

      placement=RESIDUE_CTERMINUS;

      if (p.residue.size()==0)
          placement=p.getsingleresidueplacement();

      // set placement/type for new residue.
      r.settype(residue_name,placement);
      r.setvalues(residue_chainId,
                  residue_seqNum,
                  residue_insertCode);

      // collect residue from array of atoms
      for (i=0;i<atomcnt;i++){
          try {
              r.addAtom(a[i]);
          }
          catch (char *error) {
              cerr<<"Warning; "<<error<<endl;
          }
              a[i].clear();
          }
          p.addResidue(r);
          r.clear();
      }

      return s;
  }
```

render.cc

```
// ***************************************************
// File: render.cc
// purpose: implemntation of various density field sampling methods
// for all energy types
// author: Hans-Henrik Staerfeldt
// notes:

#include "structure.hh"
#include "field.hh"
#include "stree.hh"
#include "atomiccontactenergy.hh"

//#define DEBUG

#ifdef ALTERNATIVEALGORITHM

// this is the 'improved' rendering method
void Protein::rendertreeleaf(Stree <Atom> &atomtree,
                             ReceptorGridSet &grid,
                             const int min[3],
                             const int max[3] ) {
    int x;
    int y;
    int z;
    int idx;
    static vector <Atom *> atoms;
    static double maxdist=1.0; // largest interaction
    double dist;
    double v;
    Vector3D pixel;
    int j;

#ifdef DO_Shape
    maxdist=Shape_maxradius;
#endif
#ifdef DO_Desol
    if (maxdist<Desol_maxradius)
        maxdist=Desol_maxradius;
#endif
#ifdef DO_LJfield
    if (maxdist<LJfield_maxradius)
        maxdist=LJfield_maxradius;
#endif
#ifdef DO_Elec
    if (maxdist<Elec_maxradius)
        maxdist=Elec_maxradius;
#endif

    for (x=min[0];x<=max[0];x++) {
        for (y=min[0];y<=max[0];y++) {
            idx=grid.getidx(x,y,min[2]);
            for (z=min[2];z<=max[2];z++,idx++) {
                pixel=grid.Density.getvector(x,y,z);

                atomtree.Query(pixel,maxdist,atoms);

                for (j=0;j<atoms.size();j++) {

                    dist=Vector3D::distance(pixel,atoms[j]->pos);

#ifdef DO_LJfield
                    // ******************************
                    // LJ field potential
                    // ******************************
                    static double LJfield_avgvdwradius=1.53742;

                    if (dist<LJfield_maxradius) {

                        // pos is the atom center

                        v=LJfield(grid.scale,
                                  atoms[j]->vdwradius+LJfield_avgvdwradius,
                                  atoms[j]->vdwwell,
                                  dist);
                        grid.LJfield.addmapre(idx,v);
                    }

#endif
```

```
#ifdef DO_Shape
                // ******************************
                // Shape complementarity potential
                // ******************************

                if (dist<Shape_maxradius) {
                    static const double receptor_core_factor=sqrt(1.5);
                    static const double receptor_surface_factor=sqrt(0.8);
                    static const double ligand_core_factor=sqrt(1.5);

                    // render Receptor and Ligand according to article.

                    if (atoms[j]→surface==0.0) {
                        // core atom

                        // set vwd volume exclusive field in imaginary part
                        v=Shape_rho*
                          Spherefield(grid.scale,
                                      atoms[j]→vdwradius*
                                      receptor_core_factor,
                                      dist);
                        if (v≠0.0) {
                            grid.ShapeR.setmapim(idx,v);
                            grid.ShapeR.setmapre(idx,0.0); // clear real part
                        }

                        // only set it if its not inside an atom
                        if (grid.ShapeR(idx).im==0.0) {

                            v=Spherefield(grid.scale,
                                          grid.getShape_surface()+
                                          atoms[j]→vdwradius,
                                          dist);

                            if (v≠0.0)
                              grid.ShapeR.setmapre(idx,v);
                        }

                    }else{
                        // surface atom

                        // set vwd volume exclusive field in imaginary part
                        v=Shape_rho*
                          Spherefield(grid.scale,
                                      atoms[j]→vdwradius*
                                      receptor_surface_factor,
                                      dist);
                        if (v≠0.0) {
                            grid.ShapeR.setmapim(idx,v);
                            grid.ShapeR.setmapre(idx,0.0); // clear real part
                        }

                        // only set it if its not inside an atom
                        if (grid.ShapeR(idx).im==0.0) {

                            v=Spherefield(grid.scale,
                                          grid.getShape_surface()+
                                          atoms[j]→vdwradius,
                                          dist);

                            if (v≠0.0)
                              grid.ShapeR.setmapre(idx,v);
                        }
                    }
                }
#endif

#ifdef DO_Desol
                // ******************************
                // Desolvation field grid
                // ******************************

                // render as Receptor, field
                v=Spherefield(grid.scale,
                              Desol_maxradius,
                              dist);

                // make a sum of the points
                if (v>0)
                    grid.Desolfield(idx).re+=
                      getACEfield(atoms[j]→residuetype,
                                  atoms[j]→type).re;

#endif

#ifdef DO_Elec
                // ******************************
                // Electrical field grid
                // ******************************

                // render as Receptor, field
                v=Electricalfield(grid.scale,
                                  atoms[j]→vdwradius,
                                  atoms[j]→charge,
                                  dist);

                // make a sum of the points
                grid.Elecfield.addmapre(idx,v);
                grid.Elecfield.addmapim(idx,v);

#endif
            } // loop over all atoms
        } // loop over z
    } // loop over y
  } // loop over x
}

void Protein::renderdelta(ReceptorGridSet &grid) {
  vector <Atom *> atoms;
  int j;
  int x;
  int y;
  int z;
  int idx;

  collectatoms(atoms); // collect all atoms

  for (j=0;j<atoms.size();j++) {

    x=atoms[j]→pos.x()/grid.scale;
    y=atoms[j]→pos.y()/grid.scale;
    z=atoms[j]→pos.z()/grid.scale;
    idx=grid.getidx(x,y,z);

#ifdef DO_Desol
    // ******************************
    // Desolvation density grid
    // ******************************

    grid.DesolR(idx).im+=
      getACEdensity(atoms[j]→residuetype,
                    atoms[j]→type).re;

#endif
  }
  return;
}

void Protein::rendertreerecurse(Stree <Atom> &atomtree,
                                ReceptorGridSet &grid,
                                const int min[3], const int max[3]) {
  int cmin[8][3];
  int cmax[8][3];
  static double maxdist=1.0;
  static int mincell=4;
  int sz;
  int child=0;
  Vector3D vmin;
  Vector3D vmax;
  int c[3];
  int hit[8];

#ifdef DO_Shape
  maxdist=Shape_maxradius;
#endif
#ifdef DO_Desol
  if (maxdist<Desol_maxradius)
    maxdist=Desol_maxradius;
#endif
#ifdef DO_LJfield
  if (maxdist<LJfield_maxradius)
    maxdist=LJfield_maxradius;
#endif
#ifdef DO_Elec
  if (maxdist<Elec_maxradius)
    maxdist=Elec_maxradius;
#endif

  sz=max[0]-min[0]+1;

  // only subdivide long ranges
  if (sz≤mincell) {
    rendertreeleaf(atomtree, grid, min, max);
    return;
  }

  // subdivide the index space and try again
  c[0]=ceil((min[0]+max[0])/2.0);
  c[1]=ceil((min[1]+max[1])/2.0);
```

```
    c[2]=ceil((min[2]+max[2])/2.0);

    // Child 0
    // cmin[0]=min;
    // cmax[0]=c;
    cmin[child][0]=min[0];cmin[child][1]=min[1];cmin[child][2]=min[2];
    cmax[child][0]=c[0]-1;cmax[child][1]=c[1]-1;cmax[child][2]=c[2]-1;
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 1
    // cmin[1].set(c[0],min[1],min[2]);
    // cmax[1].set(max[0],c[1],c[2]);
    cmin[child][0]=c[0];cmin[child][1]=min[1];cmin[child][2]=min[2];
    cmax[child][0]=max[0];cmax[child][1]=c[1]-1;cmax[child][2]=c[2]-1;
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 2
    // cmin[2].set(min[0],c[1],min[2]);
    // cmax[2].set(c[0],max[1],c[2]);
    cmin[child][0]=min[0];cmin[child][1]=c[1];cmin[child][2]=min[2];
    cmax[child][0]=c[0]-1;cmax[child][1]=max[1];cmax[child][2]=c[2]-1;
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 3
    // cmin[3].set(c[0],c[1],min[2]);
    // cmax[3].set(max[0],max[1],c[2]);
    cmin[child][0]=c[0];cmin[child][1]=c[1];cmin[child][2]=min[2];
    cmax[child][0]=max[0];cmax[child][1]=max[1];cmax[child][2]=c[2]-1;
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 4
    // cmin[4].set(min[0],min[1],c[2]);
    // cmax[4].set(c[0],c[1],max[2]);
    cmin[child][0]=min[0];cmin[child][1]=min[1];cmin[child][2]=c[2];
    cmax[child][0]=c[0]-1;cmax[child][1]=c[1]-1;cmax[child][2]=max[2];
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 5
    // cmin[5].set(c[0],min[1],c[2]);
    // cmax[5].set(max[0],c[1],max[2]);
    cmin[child][0]=c[0];cmin[child][1]=min[1];cmin[child][2]=c[2];
    cmax[child][0]=max[0];cmax[child][1]=c[1];cmax[child][2]=max[2];
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 6
    // cmin[6].set(min[0],c[1],c[2]);
    // cmax[6].set(c[0],max[1],max[2]);
    cmin[child][0]=min[0];cmin[child][1]=c[1];cmin[child][2]=c[2];
    cmax[child][0]=c[0]-1;cmax[child][1]=max[1];cmax[child][2]=max[2];
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 7
    // cmin[7]=c;
    // cmax[7]=max;
    cmin[child][0]=c[0];cmin[child][1]=c[1];cmin[child][2]=c[2];
    cmax[child][0]=max[0];cmax[child][1]=max[1];cmax[child][2]=max[2];
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    for (child=0;child<8;child++) {
        if (hit[child])
            rendertreerecurse(atomtree, grid, cmin[child], cmax[child]);
    }
}


// this is the improved rendering method
void Protein::rendertree(ReceptorGridSet &grid) {
    int min[3];
    int max[3];

    Stree <Atom>
      atomtree(Vector3D(0.0,0.0,0.0),
               Vector3D(grid.size*grid.scale,
                        grid.size*grid.scale,
                        grid.size*grid.scale));

    collectatoms(atomtree);

    min[0]=min[1]=min[2]=0;
    max[0]=max[1]=max[2]=grid.size-1;

    rendertreeleaf(atomtree, grid, min, max);
    renderdelta(grid);
}

// this is the improved rendering method
void Protein::rendertreefast(ReceptorGridSet &grid) {
    int min[3];
    int max[3];
    Stree <Atom>
      atomtree(Vector3D(0.0,0.0,0.0),
               Vector3D(grid.size*grid.scale,
                        grid.size*grid.scale,
                        grid.size*grid.scale));
    collectatoms(atomtree);

    min[0]=min[1]=min[2]=0;
    max[0]=max[1]=max[2]=grid.size-1;

    rendertreerecurse(atomtree, grid, min, max);
    renderdelta(grid);
}


void Protein::rendertreeleaf(Stree <Atom> &atomtree,
                             LigandGridSet &grid,
                             const int min[3],
                             const int max[3] ) {
    int x;
    int y;
    int z;
    int idx;
    static vector <Atom *> atoms;
    static double maxdist=1.0; // largest interaction
    double dist;
    double v;
    Vector3D pixel;
    int j;

#ifdef DO_Shape
    maxdist=Shape_maxradius;
#endif
#ifdef DO_Desol
    if (maxdist<Desol_maxradius)
      maxdist=Desol_maxradius;
#endif
#ifdef DO_LJfield
    if (maxdist<LJfield_maxradius)
      maxdist=LJfield_maxradius;
#endif
#ifdef DO_Elec
    if (maxdist<Elec_maxradius)
      maxdist=Elec_maxradius;
#endif

    for (x=min[0];x<=max[0];x++) {
        for (y=min[0];y<=max[0];y++) {
            idx=grid.getidx(x,y,min[2]);
            for (z=min[2];z<=max[2];z++,idx++) {
                pixel=grid.Density.getvector(x,y,z);

                atomtree.Query(pixel,maxdist,atoms);

                for (j=0;j<atoms.size();j++) {

                    dist=Vector3D::distance(pixel,atoms[j]->pos);


#ifdef DO_Shape
                    // ******************************
                    // Shape complementarity potential
                    // ******************************

                    if (dist<Shape_maxradius) {
                        static const double receptor_core_factor=sqrt(1.5);
                        static const double receptor_surface_factor=sqrt(0.8);
                        static const double ligand_core_factor=sqrt(1.5);

                        // render Receptor and Ligand according to article.

                        if (atoms[j]->surface==0.0) {
                            // core atom

                            if (grid.ShapeL(idx).re==0.0) {
```

```
                        v=Shape_rho*
                          Spherefield(grid.scale,
                                      atoms[j]->vdwradius*
                                      ligand_core_factor,
                                      dist);
                        if (v)
                          grid.ShapeL.setmapim(idx,v);
                     }

                  }else{
                     // surface atom

                     // assign 1 to surface atoms in ligand
                     v=Spherefield(grid.scale,
                                   atoms[j]->vdwradius,
                                   dist);
                     if (v) {
                        grid.ShapeL.setmapre(idx,v);
                        grid.ShapeL.setmapim(idx,0);
                     }
                  }
               }
#endif

#ifdef DO_Desol
               // ******************************
               // Desolvation field grid
               // ******************************

               // render as Receptor, field
               v=Spherefield(grid.scale,
                             Desol_maxradius,
                             dist);

               // make a sum of the points
               if (v>0)
                 grid.Desold(idx).re+=
                   getACEfield(atoms[j]->residuetype,
                               atoms[j]->type).re;

#endif


            } // loop over all atoms
         } // loop over z
      } // loop over y
   } // loop over x
}

void Protein::renderdelta(LigandGridSet &grid) {
   vector <Atom *> atoms;
   int j;
   int x;
   int y;
   int z;
   int idx;

   collectatoms(atoms); // collect all atoms

   for (j=0;j<atoms.size();j++) {

      x=atoms[j]->pos.x()/grid.scale;
      y=atoms[j]->pos.y()/grid.scale;
      z=atoms[j]->pos.z()/grid.scale;
      idx=grid.getidx(x,y,z);

#ifdef DO_Elec
      // ******************************
      // Electrical density grid
      // ******************************
      if (atoms[j]->charge<0) {
         grid.Elecdm(idx).im+=(-atoms[j]->charge);
         grid.Elecdl(idx).im+=(-atoms[j]->charge);
      }else{
         grid.Elecdm(idx).re+=atoms[j]->charge;
         grid.Elecdl(idx).re+=atoms[j]->charge;
      }
#endif


#ifdef DO_Desol
      // ******************************
      // Desolvation density grid
      // ******************************

      grid.DesolL(idx).im+=
        getACEdensity(atoms[j]->residuetype,
                      atoms[j]->type).re;

#endif
      // ******************************
      // Density delta function (possible co-occupation oc course grid)
      // ******************************

      grid.Density(idx).re+=1.0;

   }// loop over collected atoms
   atoms.clear();
}

void Protein::rendertreerecurse(Stree <Atom> &atomtree,
                                LigandGridSet &grid,
                                const int min[3],
                                const int max[3]) {
   int cmin[8][3];
   int cmax[8][3];
   static double maxdist=1.0;
   static int mincell=4;
   int sz;
   int child=0;
   Vector3D vmin;
   Vector3D vmax;
   int c[3];
   int hit[8];

#ifdef DO_Shape
   maxdist=Shape_maxradius;
#endif
#ifdef DO_Desol
   if (maxdist<Desol_maxradius)
     maxdist=Desol_maxradius;
#endif
#ifdef DO_LJfield
   if (maxdist<LJfield_maxradius)
     maxdist=LJfield_maxradius;
#endif
#ifdef DO_Elec
   if (maxdist<Elec_maxradius)
     maxdist=Elec_maxradius;
#endif

   sz=max[0]-min[0]+1;

   // only subdivide long ranges
   if (sz<=mincell) {
      rendertreeleaf(atomtree, grid, min, max);
      return;
   }

   // subdivide the index space and try again
   c[0]=ceil((min[0]+max[0])/2.0);
   c[1]=ceil((min[1]+max[1])/2.0);
   c[2]=ceil((min[2]+max[2])/2.0);

   // Child 0
   // cmin[0]=min;
   // cmax[0]=c;
   cmin[child][0]=min[0];cmin[child][1]=min[1];cmin[child][2]=min[2];
   cmax[child][0]=c[0]-1;cmax[child][1]=c[1]-1;cmax[child][2]=c[2]-1;
   vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
   vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
   hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

   // Child 1
   // cmin[1].set(c[0],min[1],min[2]);
   // cmax[1].set(max[0],c[1],c[2]);
   cmin[child][0]=c[0];cmin[child][1]=min[1];cmin[child][2]=min[2];
   cmax[child][0]=max[0];cmax[child][1]=c[1]-1;cmax[child][2]=c[2]-1;
   vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
   vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
   hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

   // Child 2
   // cmin[2].set(min[0],c[1],min[2]);
   // cmax[2].set(c[0],max[1],c[2]);
   cmin[child][0]=min[0];cmin[child][1]=c[1];cmin[child][2]=min[2];
   cmax[child][0]=c[0]-1;cmax[child][1]=max[1];cmax[child][2]=c[2]-1;
   vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
   vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
   hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

   // Child 3
   // cmin[3].set(c[0],c[1],min[2]);
   // cmax[3].set(max[0],max[1],c[2]);
   cmin[child][0]=c[0];cmin[child][1]=c[1];cmin[child][2]=min[2];
   cmax[child][0]=max[0];cmax[child][1]=max[1];cmax[child][2]=c[2]-1;
   vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
   vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
   hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);
```

```
    // Child 4
    // cmin[4].set(min[0],min[1],c[2]);
    // cmax[4].set(c[0],c[1],max[2]);
    cmin[child][0]=min[0];cmin[child][1]=min[1];cmin[child][2]=c[2];
    cmax[child][0]=c[0]-1;cmax[child][1]=c[1]-1;cmax[child][2]=max[2];
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 5
    // cmin[5].set(c[0],min[1],c[2]);
    // cmax[5].set(max[0],c[1],max[2]);
    cmin[child][0]=c[0];cmin[child][1]=min[1];cmin[child][2]=c[2];
    cmax[child][0]=max[0];cmax[child][1]=c[1];cmax[child][2]=max[2];
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 6
    // cmin[6].set(min[0],c[1],c[2]);
    // cmax[6].set(c[0],max[1],max[2]);
    cmin[child][0]=min[0];cmin[child][1]=c[1];cmin[child][2]=c[2];
    cmax[child][0]=c[0]-1;cmax[child][1]=max[1];cmax[child][2]=max[2];
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    // Child 7
    // cmin[7]=c;
    // cmax[7]=max;
    cmin[child][0]=c[0];cmin[child][1]=c[1];cmin[child][2]=c[2];
    cmax[child][0]=max[0];cmax[child][1]=max[1];cmax[child][2]=max[2];
    vmin=grid.getvector(cmin[child][0],cmin[child][1],cmin[child][2]);
    vmax=grid.getvector(cmax[child][0],cmax[child][1],cmax[child][2]);
    hit[child++]=atomtree.Querybool(vmin,vmax,maxdist);

    for (child=0;child<8;child++) {
        if (hit[child])
            rendertreerecurse(atomtree, grid, cmin[child], cmax[child]);
    }
}


// this is the improved rendering method
void Protein::rendertree(LigandGridSet &grid) {
    int min[3];
    int max[3];

    Stree <Atom>
        atomtree(Vector3D(0.0,0.0,0.0),
                 Vector3D(grid.size*grid.scale,
                          grid.size*grid.scale,
                          grid.size*grid.scale));
    collectatoms(atomtree);

    min[0]=min[1]=min[2]=0;
    max[0]=max[1]=max[2]=grid.size-1;

    rendertreeleaf(atomtree, grid, min, max);
    renderdelta(grid);
}

// this is the improved rendering method
void Protein::rendertreefast(LigandGridSet &grid) {
    int min[3];
    int max[3];
    Stree <Atom>
        atomtree(Vector3D(0.0,0.0,0.0),
                Vector3D(grid.size*grid.scale,
                         grid.size*grid.scale,
                         grid.size*grid.scale));
    collectatoms(atomtree);

    min[0]=min[1]=min[2]=0;
    max[0]=max[1]=max[2]=grid.size-1;

    rendertreerecurse(atomtree, grid, min, max);
    renderdelta(grid);
}

#endif


void Protein::render(LigandGridSet &grid) {
    int i;

#ifdef DO_Shape

    int a,j,k;
    int neighbouridx[26];

    // make fast index of neighbour offsets
    a=0;
    for (i=-1;i<2;i++) {
        for (j=-1;j<2;j++) {
            for (k=-1;k<2;k++) {
                if (i==0 && j==0 && k==0) continue;
                neighbouridx[a++]=grid.getidx(i,j,k);
            }
        }
    }
    assert(a==26); // wooo!
#endif

    findsurface(1.7); // make surface

    for (i=0;i<residue.size();i++) {
#ifdef DEBUG2
        cerr<<ResidueInfoData[residue[i].type].key<<",";
#endif
        residue[i].render(grid);
    }
#ifdef DEBUG2
    cerr<<"."<<endl;
#endif

    // adjust shape complentarity fields for coocupation with atoms
    for (i=0;i<grid.sizecubed;i++) {

#ifdef DO_Shape
        // change according to atricle
        if (grid.ShapeL(i).im!=0.0) {
            for (a=0;a<26;a++) {
                int nidx=i+neighbouridx[a];
                if (grid.ShapeL(nidx).re==0.0)
                    grid.ShapeL(nidx).re=1.0;
            }
        }
#endif
    }
}

void Protein::render(ReceptorGridSet &grid) {
    int i;

    findsurface(1.7); // make surface

    for (i=0;i<residue.size();i++) {
#ifdef DEBUG2
        cerr<<ResidueInfoData[residue[i].type].key<<",";
#endif
        residue[i].render(grid);
    }

#ifdef DEBUG2
    cerr<<"."<<endl;
#endif
}

void Residue::render(ReceptorGridSet &grid) {
    int i;
    for (i=0;i<size();i++) {
        if (atom[i].type==RATOM_MAX) continue; // skip if not there
        // todo: determine rendering plane and field type
        atom[i].render(grid);
    }
}

void Residue::render(LigandGridSet &grid) {
    int i;
    for (i=0;i<size();i++) {
        if (atom[i].type==RATOM_MAX) continue; // skip if not there
        // todo: determine rendering plane and field type
        atom[i].render(grid);
    }
}

void Atom::render(ReceptorGridSet &grids) {

    // get radius
    int x,y,z;
    int pixelwidth;
    int min[3];
    int max[3];
    Vector3D pixel; // real position of pixel
    int idx;
    double v;
    double maxdist;
    double dist;
```

```
#ifdef DO_Shape
   maxdist=Shape_maxradius;
#endif
#ifdef DO_Desol
   if (maxdist<Desol_maxradius)
     maxdist=Desol_maxradius;
#endif
#ifdef DO_LJfield
   if (maxdist<LJfield_maxradius)
     maxdist=LJfield_maxradius;
#endif
#ifdef DO_Elec
   if (maxdist<Elec_maxradius)
     maxdist=Elec_maxradius;
#endif

   // get pixel width
   pixelwidth=ceil(maxdist/grids.scale); // get pixel width

   // get the clipped box of the ljfield
   grids.getclippedbox(pos, pixelwidth, min, max);

   for (x=min[0];x<=max[0];x++){
      for (y=min[1];y<=max[1];y++){
         idx=grids.getidx(x,y,min[2]);
         for (z=min[2];z<=max[2];z++,idx++){

            pixel=grids.getvector(x,y,z); // get pixel position

            // distance between atom position and pixel position
            dist=Vector3D::distance(pos,pixel);

#ifdef DO_LJfield
            // ******************************
            // LJ field potential
            // ******************************

            if (dist < LJfield_maxradius ){
               // pos is the atom center
               static const double LJfield_avgvdwradius=1.53742;

               v=LJfield(grids.scale,
                         vdwradius+LJfield_avgvdwradius,
                         vdwwell,dist);
               grids.LJfield.addmapre(idx,v);
            }
#endif

#ifdef DO_Shape
            // ******************************
            // Shape complementarity potential
            // ******************************

            if (dist<Shape_maxradius) {

               static const double receptor_core_factor=sqrt(1.5);
               static const double receptor_surface_factor=sqrt(0.8);
               static const double ligand_core_factor=sqrt(1.5);

               if (surface==-1)
                 throw "Surface not calculated in rendering atom";

               // render Receptor and Ligand according to article.

               if (surface==0.0) {

                  // ************************
                  // render receptor core grid

                  // set vwd volume exclusive field in imaginary part
                  v=Shape_rho*
                    Spherefield(grids.scale,
                                vdwradius*receptor_core_factor,
                                dist);
                  if (v≠0.0) {
                     grids.ShapeR.setmapim(idx,v);
                     grids.ShapeR.setmapre(idx,0.0); // clear real part
                  }

                  // only set it if its not inside an atom
                  if (grids.ShapeR(idx).im==0.0) {

                     v=Spherefield(grids.scale,
                                   Shape_surface+vdwradius,
                                   dist);
                     if (v≠0.0)
                       grids.ShapeR.setmapre(idx,v);
                  }

               }else{

                  // ************************
                  // render receptor surface grid

                  // set vwd volume exclusive field in imaginary part
#ifdef NOMODIFICATIONODSHAPE
                  v=Shape_rho*
                    Spherefield(grids.scale,
                                vdwradius*receptor_surface_factor,
                                dist);
                  if (v≠0.0) {
                     grids.ShapeR.setmapim(idx,v);
                     grids.ShapeR.setmapre(idx,0.0); // clear real part
                  }
#endif

                  // only set it if its not inside an atom
                  if (grids.ShapeR(idx).im==0.0) {

                     v=Spherefield(grids.scale,
                                   Shape_surface+vdwradius,
                                   dist);
                     if (v≠0.0)
                       grids.ShapeR.setmapre(idx,v);
                  }
               }
            }

#endif

#ifdef DO_Elec
            // ******************************
            // Electrical field grids
            // ******************************

            // get radius r
            if (dist<=Elec_maxradius) {

               // render as Receptor, field
               v=Electricalfield(grids.scale,
                                 vdwradius,
                                 charge,
                                 dist);

               // make a sum of the points
               grids.Elecfield.addmapre(idx,v);
               grids.Elecfield.addmapim(idx,v);
            }
#endif

#ifdef DO_Desol
            // ******************************
            // Desolvation field grid
            // ******************************

            // render as Receptor, field
            v=Spherefield(grids.scale,
                          Desol_maxradius,
                          dist);

            // make a sum of the points
            if (v>0)
              grids.DesolR(idx).re=
                grids.DesolR(idx).re+
                  getACEfield(residuetype,type).re;

#endif

         } // loop over box z
      } // loop over box y
   } // loop over box x

}


void Atom::render(LigandGridSet &grids) {

   // get radius
   int x,y,z;
   int pixelwidth;
   int min[3];
   int max[3];
   Vector3D pixel; // real position of pixel
   int idx;
   double v;
   double maxdist;
   double dist;

#ifdef DO_Shape
```

```
    maxdist=Shape_maxradius;

    // get pixel width
    pixelwidth=ceil(maxdist/grids.scale); // get pixel width

    // get the clipped box of the ljfield
    grids.getclippedbox(pos, pixelwidth, min, max);

    for (x=min[0];x≤max[0];x++){
        for (y=min[1];y≤max[1];y++){
            idx=grids.getidx(x,y,min[2]);
            for (z=min[2];z≤max[2];z++,idx++){

                pixel=grids.getvector(x,y,z); // get pixel position

                // distance between atom position and pixel position
                dist=Vector3D::distance(pos,pixel);

                // ******************************
                // Shape complementarity potential
                // ******************************

                if (dist<Shape_maxradius) {

                    static const double receptor_core_factor=sqrt(1.5);
                    static const double receptor_surface_factor=sqrt(0.8);
                    static const double ligand_core_factor=sqrt(1.5);

                    if (surface==-1)
                      throw "Surface not calculated in rendering atom";

                    // render Receptor and Ligand according to article.

                    if (surface==0.0) {

                        // ************************
                        // render ligand core grid
                        // assign rho to core atoms in ligand (imag part),
                        // only if they dont overlap with surface atoms (real part)
                        if (grids.ShapeL(idx).re==0.0) {
                            v=Shape_rho*
                              Spherefield(grids.scale,
                                          vdwradius*ligand_core_factor,
                                          dist);
                            if (v≠0.0)
                              grids.ShapeL.setmapim(idx,v);
                        }

                    }else{


                        // ************************
                        // render ligand surface grid

                        // assign 1 to surface atoms in ligand
                        v=Spherefield(grids.scale,
                                      vdwradius,
                                      dist);
                        if (v≠0.0) {
                            grids.ShapeL.setmapre(idx,v);
                            grids.ShapeL.setmapim(idx,0.0); // clear core
                        }
                    }
                }

            } // loop over box z
        } // loop over box y
    } // loop over box x

#endif

    x=pos.x()/grids.scale;
    y=pos.y()/grids.scale;
    z=pos.z()/grids.scale;

#ifdef DO_Desol

    // render as Ligand, density
    grids.DesolL(x,y,z).im+=
      getACEdensity(residuetype,type).re;

#endif

#ifdef DO_Elec

    // render as Ligand, density
    grids.Elecd(idx).re+=charge;

#endif

    // ******************************
    // Density delta function (possible co-occupation of course grid)
    // ******************************
    grids.Density(x,y,z).re+=1.0;

}


void Protein::findsurface(double radius) {
    CGrid surfacegrid;
    vector <Atom *> atoms;
    Stree <Atom> *atomtree;
    Vector3D cog;
    double diameter;
    double scale=0.6;
    int size;
    double surfaceprcell=0.01; // as scale is constant, so should this
    int i,j,k;
    int x,y,z;
    int idx;
    int min[3];
    int max[3];
    Vector3D bmin;
    Vector3D bmax;
    Vector3D pixel;
    Vector3D grid;
    double pixelwidth;
    double dist;
    double maxvdw=2.2;
    int neighbouridx[26];
    const double safety=1.4;
    int r;
    int a;

    // ******************************
    // surface approximation function
    // ******************************

    if (surfaceball==radius) return;
    cerr≪"find surface with ball at "≪radius≪endl;
    cog=getcog();
    diameter=getdiameter(cog);

    // have a constant scale, an vary the grid size to fit
    size=(safety*diameter+2*radius)/scale;
    surfacegrid.init(scale,size);

    bmin.set(-diameter*safety,-diameter*safety,-diameter*safety);
    bmax.set(diameter*safety,diameter*safety,diameter*safety);
    bmin+=cog;
    bmax+=cog;
    atomtree=new Stree <Atom> (bmin,bmax);

    collectatoms(atoms);

    // find translation vector (but dont translate)
    grid.set(size*scale/2.0,size*scale/2.0,size*scale/2.0);
    grid-=cog;

    // for each atom, draw a sphere around
    // it at a radius+vdwradius radius.
    for (a=0;a<atoms.size();a++) {

        // get pixel width
        pixelwidth=ceil((atoms[a]→vdwradius+radius)/surfacegrid.getscale());

        // get the clipped box of the field
        surfacegrid.getclippedbox(atoms[a]→pos+grid, pixelwidth, min, max);

        // mark that the surface has been calculated
        atoms[a]→surface=0;

        for (x=min[0];x≤max[0];x++){
            for (y=min[1];y≤max[1];y++){
                idx=surfacegrid.getidx(x,y,min[2]);
                for (z=min[2];z≤max[2];z++,idx++){

                    pixel=surfacegrid.getvector(x,y,z); // get pixel position

                    // distance between atom position and pixel position
                    dist=Vector3D::distance(atoms[a]→pos+grid,pixel);

                    if (dist<(atoms[a]→vdwradius+radius)) {
                        surfacegrid(idx).re=1;
                    }
                }
            }
        }
    }
    // tested this, it works, only thing is the
```

```
    // grid is a bit bigger than needed (which is ok)

#ifdef SAVETEST
 {
     ofstream of;
     of.open("data/surfacetest.grid");
     of<<surfacegrid;
     of.close();
     exit(1);
 }
#endif

    collectatoms(*atomtree);

    // make fast index of neighbour offsets
    a=0;
    for (i=-1;i<2;i++) {
        for (j=-1;j<2;j++) {
            for (k=-1;k<2;k++) {
                if (i==0 && j==0 && k==0) continue;
                neighbouridx[a++]=surfacegrid.getidx(i,j,k);
            }
        }
    }
    assert(a==26); // wooo!

    // loop through all colored cells in the grid.
    for (x=0,idx=0;x<size;x++){
        for (y=0;y<size;y++){
            for (z=0;z<size;z++,idx++){

                if (x==0) continue;
                if (y==0) continue;
                if (z==0) continue;

                if (x==size-1) continue;
                if (y==size-1) continue;
                if (z==size-1) continue;

                if (surfacegrid(idx).re!=1) continue; // its not a surface atom

                pixel=surfacegrid.getvector(x,y,z);

                // if a neighbour cell is uncolored, add surface to nearby atoms.
                atoms.clear();

                // loop though neighbours
                // this has been tested ok.
                for (a=0;a<26;a++) {
                    int nidx=idx+neighbouridx[a];

                    if (surfacegrid(nidx).re==0) {
                        (*atomtree).Query(pixel-grid,radius+maxvdw,atoms);
                        break;
                    }
                }

                // see if we found an edge cell and any atoms
                for (a=0;a<atoms.size();a++){
                    // add the contribution to the atoms
                    // within the surface cells radius
                    atoms[a]->surface+=surfaceprcell;
                    if (atoms[a]->surface>9.0) atoms[a]->surface=9.0;
                }
            }
        }
    }

    // sum up surface for residues
    for (r=0;r<residue.size();r++) {
        residue[r].surface=0;
        for (a=0;a<residue[r].size();a++){
            if (residue[r].atom[a].type==RATOM_MAX) continue;
            residue[r].surface+=residue[r].atom[a].surface;
        }
    }

    delete atomtree;
    surfaceball=radius;
}
```

build.cc

```
// ************************************************
// File: build.cc
// purpose: functions that build both hydrogen atoms
// and sidechain atoms from rotamer parameters
// for all residue types.
// author: Hans-Henrik Staerfeldt
// notes:

#include "structure.hh"
#include "vector3d.hh"

#define CHECK

// Adds an atom of type 'rtype' tp the residue, using
// a relative position 'rpos' to three reference atoms
// 'c1,c2,c3'.
void Residue::addAtom(const ResidueAtomtype rtype,
                      const double bond,
                      const double theta,
                      const double phi,
                      const ResidueAtomtype c1,
                      const ResidueAtomtype c2,
                      const ResidueAtomtype c3) {
    Matrix4D iM;
    int atomnum;
    Vector3D rpos;
    static char error[256];

    ResidueInfo *ResidueInfoDataPtr;

    switch (placement) {
     case RESIDUE_CTERMINUS:
        ResidueInfoDataPtr=ResidueInfoDatact;
        break;
     case RESIDUE_CENTER:
        ResidueInfoDataPtr=ResidueInfoData;
        break;
     case RESIDUE_NTERMINUS:
        ResidueInfoDataPtr=ResidueInfoDatant;
        break;
    }

    atomnum=ResidueInfoDataPtr[type].atomnum[rtype];
    if (atomnum==-1 && ResidueInfoDataPtr[type].atomcount==0) {
        sprintf(error,"Warning; could not build residue %s, "
                "residue is not defined for %s",
                ResidueInfoDataPtr[type].key,
                PlacementInfoData[placement]);
        throw error;
    }

    atom[atomnum].vdwradius=ResidueInfoDataPtr[type].vdwradius[atomnum];
    atom[atomnum].vdwwell=ResidueInfoDataPtr[type].vdwwell[atomnum];
    atom[atomnum].charge=ResidueInfoDataPtr[type].charge[atomnum];

#ifdef CHECK
    if ((*this)[c1].type==RATOM_MAX) {
        sprintf(error,"Warning; could not build atom %s to "
                "residue %s(%d), atom %s is not yet defined",
                AtomInfoData[rtype].key, ResidueInfoDataPtr[type].key,
                seqNum, AtomInfoData[c1].key);

        cerr<<error<<endl;
        return;
// throw error;
    }
    if ((*this)[c2].type==RATOM_MAX) {
        sprintf(error,"Warning; could not build atom %s to "
                "residue %s(%d), atom %s is not yet defined",
                AtomInfoData[rtype].key, ResidueInfoDataPtr[type].key,
                seqNum, AtomInfoData[c2].key);
        cerr<<error<<endl;
// throw error;
    }
    if ((*this)[c3].type==RATOM_MAX) {
        sprintf(error,"Warning; could not build atom %s to "
                "residue %s(%d), atom %s is not yet defined",
                AtomInfoData[rtype].key, ResidueInfoDataPtr[type].key,
                seqNum, AtomInfoData[c3].key);
        cerr<<error<<endl;
        return;
// throw error;
    }
#endif

    if (atomnum==-1) {
        sprintf(error,"Warning; could not build illegal atom %s of "
                "residue %s (%s)",
                AtomInfoData[rtype].key, ResidueInfoDataPtr[type].key,
                PlacementInfoData[placement]);
        cerr<<error<<endl;
        return;
// throw error;
    }

    atom[atomnum].type=rtype;
```

```
    rpos.set(bond*sin(theta)*sin(phi),
             bond*sin(theta)*cos(phi),
             bond*cos(theta));

    iM.iviewat((*this)[c1].pos,(*this)[c2].pos,(*this)[c3].pos);
    atom[atomnum].pos=rpos*iM;
}

void Protein::buildhydrogen(void) {
    int i;
    double phi;                          /* C(i-1) - N(i) - CA(i) - C(i) */
    double psi;                          /* N(i) - CA(i) - C(i) - N(i+1) */
    char chainId;


    if (residue.size()==0) return; // all done :-)
    phi = 0;
    if (residue.size()>1)
      psi = Vector3D::dihedral(residue[0][RATOM_N].pos,
                               residue[0][RATOM_CA].pos,
                               residue[0][RATOM_C].pos,
                               residue[1][RATOM_N].pos);
    else
      psi=0; // if we have only one residue

    residue[0].buildhydrogen(phi,psi);

    for (i=1;i<residue.size();i++){

        chainId=residue[i].chainId;

        if (residue[i-1].chainId==chainId)
          phi = Vector3D::dihedral(residue[i-1][RATOM_C].pos,
                                   residue[i][RATOM_N].pos,
                                   residue[i][RATOM_CA].pos,
                                   residue[i][RATOM_C].pos);
        else // start of chain
          phi=0;

        if (i<residue.size()-1 && residue[i+1].chainId==chainId)
          psi = Vector3D::dihedral(residue[i][RATOM_N].pos,
                                   residue[i][RATOM_CA].pos,
                                   residue[i][RATOM_C].pos,
                                   residue[i+1][RATOM_N].pos);
        else
          psi=0; // last one, or end of chain

        residue[i].buildhydrogen(phi,psi);
  }

}

// build residue using backbone positions and chi angles
void Residue::buildhydrogen(const double bb_phi, const double bb_psi) {
    double bond;
    double theta;
    double phi;
    Vector3D v;

    // Note : In this generation, CA,N,C is assumed to be fixed, and all
    // gerenation is done in reference to these atoms.

    switch (type) {

     case RESIDUE_MAX:
        // say what?
        throw "Illegal residue type (build)";
        break;

     case RESIDUE_ALA:

        if (placement==RESIDUE_NTERMINUS) {
            /* ALA H1 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.5);
            phi=degreetoradian(60);
            addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

            /* ALA H2 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.5);
            phi=degreetoradian(180);
            addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

            /* ALA H3 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.5);
            phi=degreetoradian(300);
            addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

         }else{
            /* ALA H */
            /* bond: amber */
            /* angle: amber */
            /* The placement of the HN depends on the phi angle,
               as its in plane with the previous C.
               modified for CA-CB reference phi;360-119.800 */
            bond=1.010;
            theta=degreetoradian(240.2);
            phi=degreetoradian(bb_phi-120);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
            /* Original entry;
               theta=degreetoradian(119.800);
               phi=degreetoradian(0.000);
               addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
               */
        }

        /* ALA HA */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(120.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
        /* Original entry;
            bond=1.090;
            theta=degreetoradian(109.500);
            phi=degreetoradian(300.000);
            addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
            */

        /* ALA HB1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_HB1,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

        /* ALA HB2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

        /* ALA HB3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

        break;

      case RESIDUE_ARG:

        if (placement==RESIDUE_NTERMINUS) {
            /* ARG H1 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.5);
            phi=degreetoradian(60);
            addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

            /* ARG H2 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.5);
            phi=degreetoradian(180);
            addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

            /* ARG H3 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.5);
            phi=degreetoradian(300);
```

```
    addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

}else{
    /* ARG H */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the HN depends on the phi angle,
        as its in plane with the previous C.
        modified for CA-CB reference phi;360-119.800 */
    bond=1.010;
    theta=degreetoradian(240.2);
    phi=degreetoradian(bb_phi-120);
    addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
    /* Original entry;
        theta=degreetoradian(119.800);
        phi=degreetoradian(0.000);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
        */
}

/* ARG HA */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(120.000);
addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
/* Original entry;
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
    */

/* ARG HB2 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(300.000);
phi=degreetoradian(240.000);
addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

/* ARG HB3 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(60.000);
phi=degreetoradian(120.000);
addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

/* ARG HG2 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(240.000);
addAtom(RATOM_HG2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CD);

/* ARG HG3 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(120.000);
addAtom(RATOM_HG3,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CD);

/* ARG HD2 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(240);
addAtom(RATOM_HD2,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_NE);

/* ARG HD3 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(120);
addAtom(RATOM_HD3,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_NE);

/* ARG HE */
/* bond: amber */
/* angle: amber */
bond=1.010;
theta=degreetoradian(118.500);
phi=degreetoradian(180);
addAtom(RATOM_HE,bond,theta,phi,RATOM_NE,RATOM_CD,RATOM_CZ);

/* ARG HH11 */
/* bond: amber */
/* angle: amber */
bond=1.010;
theta=degreetoradian(119.800);
phi=degreetoradian(0.000);
addAtom(RATOM_HH11,bond,theta,phi,RATOM_NH1,RATOM_CZ,RATOM_NE);

/* ARG HH12 */
/* bond: amber */
/* angle: amber */
bond=1.010;
theta=degreetoradian(119.800);
phi=degreetoradian(180.000);
addAtom(RATOM_HH12,bond,theta,phi,RATOM_NH1,RATOM_CZ,RATOM_NE);

/* ARG HH21 */
/* bond: amber */
/* angle: amber */
bond=1.010;
theta=degreetoradian(119.800);
phi=degreetoradian(0.000);
addAtom(RATOM_HH21,bond,theta,phi,RATOM_NH2,RATOM_CZ,RATOM_NE);

/* ARG HH22 */
/* bond: amber */
/* angle: amber */
bond=1.010;
theta=degreetoradian(119.800);
phi=degreetoradian(180.000);
addAtom(RATOM_HH22,bond,theta,phi,RATOM_NH2,RATOM_CZ,RATOM_NE);

break;

case RESIDUE_ASN:

if (placement==RESIDUE_NTERMINUS) {
    /* ASN H1 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.5);
    phi=degreetoradian(60.000);
    addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    /* ASN H2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.500);
    phi=degreetoradian(180.000);
    addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    /* ASN H3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

}else{
    /* ASN H */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the HN depends on the phi angle,
        as its in plane with the previous C.
        modified for CA-CB reference phi;360-119.800 */
    bond=1.010;
    theta=degreetoradian(240.2);
    phi=degreetoradian(bb_phi-120);
    addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
    /* Original entry;
        theta=degreetoradian(119.800);
        phi=degreetoradian(0.000);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
        */
}

/* ASN HA */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(120.000);
addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
/* Original entry;
    bond=1.090;
```

```
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* ASN HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* ASN HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* ASN HD21 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(119.800);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HD21,bond,theta,phi,RATOM_ND2,RATOM_CG,RATOM_CB);

    /* ASN HD22 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(119.800);
    phi=degreetoradian(0.000);
    addAtom(RATOM_HD22,bond,theta,phi,RATOM_ND2,RATOM_CG,RATOM_CB);

    break;

case RESIDUE_ASP:
case RESIDUE_ASH:

    if (placement==RESIDUE_NTERMINUS) {
        /* ASP H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* ASP H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* ASP H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* ASP,ASH H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* ASP,ASH HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* ASP,ASH HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* ASP,ASH HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    if (type==RESIDUE_ASH) {
        /* ASH HD2 */
        /* bond: amber */
        /* angle: amber */
        bond=0.96;
        theta=degreetoradian(109.5);
        phi=degreetoradian(180.0);
        addAtom(RATOM_HD2,bond,theta,phi,RATOM_OD2,RATOM_CG,RATOM_CB);
    }

    break;

case RESIDUE_CYS:
case RESIDUE_CYM:
case RESIDUE_CYX:

    if (placement==RESIDUE_NTERMINUS) {
        /* CYS H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* CYS H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* CYS H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* CYS,CYM,CYX H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
        modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        if (type==RESIDUE_CYM)
          addAtom(RATOM_HN,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        else
          addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            addAtom(RATOM_HN,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
```

```
        }

        /* CYS,CYM,CYX HA */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(120.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
        /* Original entry;
            bond=1.090;
            theta=degreetoradian(109.500);
            phi=degreetoradian(300.000);
            addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
            */

        /* CYS,CYM,CYX HB2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(240.000);
        addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_SG);

        /* CYS,CYM,CYX HB3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(120.000);
        addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_SG);

        if (type==RESIDUE_CYS) {
            /* CYS HG */
            /* bond: amber */
            /* angle: amber */
            bond=1.330;
            theta=degreetoradian(96.000);
            phi=degreetoradian(180.000);
            if (placement==RESIDUE_NTERMINUS ||
placement==RESIDUE_CTERMINUS)
                addAtom(RATOM_HSG,bond,theta,phi,RATOM_SG,RATOM_CB,RATOM_CA);
            else
                addAtom(RATOM_HG,bond,theta,phi,RATOM_SG,RATOM_CB,RATOM_CA);
        }

        break;

    case RESIDUE_GLN:

        if (placement==RESIDUE_NTERMINUS) {
            /* GLN H1 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.500);
            phi=degreetoradian(60.000);
            addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

            /* GLN H2 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.500);
            phi=degreetoradian(180.000);
            addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

            /* GLN H3 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.500);
            phi=degreetoradian(300.000);
            addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        }else{
            /* GLN H */
            /* bond: amber */
            /* angle: amber */
            /* The placement of the HN depends on the phi angle,
                as its in plane with the previous C.
                modified for CA-CB reference phi;360-119.800 */
            bond=1.010;
            theta=degreetoradian(240.2);
            phi=degreetoradian(bb_phi-120);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
            /* Original entry;
                theta=degreetoradian(119.800);
                phi=degreetoradian(0.000);
                addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
                */
        }

        /* GLN HA */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(120.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
        /* Original entry;
            bond=1.090;
            theta=degreetoradian(109.500);
            phi=degreetoradian(300.000);
            addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
            */

        /* GLN HB2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(120.000);
        addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

        /* GLN HB3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(240.000);
        addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

        /* GLN HG2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HG2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

        /* GLN HG3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_HG3,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

        /* GLN HE21 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(119.800);
        phi=degreetoradian(180.000);
        addAtom(RATOM_HE21,bond,theta,phi,RATOM_NE2,RATOM_CD,RATOM_CG);

        /* GLN HE22 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(119.800);
        phi=degreetoradian(0.000);
        addAtom(RATOM_HE22,bond,theta,phi,RATOM_NE2,RATOM_CD,RATOM_CG);

        break;

    case RESIDUE_GLU:
    case RESIDUE_GLH:

        if (placement==RESIDUE_NTERMINUS) {
            /* GLU H1 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.500);
            phi=degreetoradian(60.000);
            addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

            /* GLU H2 */
            /* bond: amber */
            /* angle: amber */
            bond=1.010;
            theta=degreetoradian(109.500);
            phi=degreetoradian(180.000);
            addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

            /* GLU H3 */
            /* bond: amber */
            /* angle: amber */
```

```
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* GLU,GLH H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* GLU,GLH HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* GLU,GLH HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* GLU,GLH HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* GLU,GLH HG2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HG2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CD);

    /* GLU,GLH HG3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HG3,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CD);

    if (type==RESIDUE_GLH) {
        /* GLH HE2 */
        /* bond: amber */
        /* angle: amber */
        bond=0.960;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_HE2,bond,theta,phi,RATOM_OE2,RATOM_CD,RATOM_CG);
    }

    break;

case RESIDUE_GLY:

    if (placement==RESIDUE_NTERMINUS) {
        /* GLY H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* GLY H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* GLY H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* GLY H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2-120+120);
        phi=degreetoradian(bb_phi);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* GLY HA2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA2,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* GLY HA3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HA3,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    break;

case RESIDUE_HIS:
case RESIDUE_HID:
case RESIDUE_HIE:
case RESIDUE_HIP:

    if (placement==RESIDUE_NTERMINUS) {
        /* HIS,HID,HIE,HIP H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* HIS,HID,HIE,HIP H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* HIS,HID,HIE,HIP H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* HIS,HID,HIE,HIP H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
```

```
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* HIS,HID,HIE,HIP HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* HIS,HID,HIE,HIP HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* HIS,HID,HIE,HIP HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    if (type==RESIDUE_HIS || type==RESIDUE_HID || type==RESIDUE_HIP) {
        /* HIS,HID,HIP HD1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(126.000);
        phi=degreetoradian(0.000);
        addAtom(RATOM_HD1,bond,theta,phi,RATOM_ND1,RATOM_CG,RATOM_CB);
    }

    /* HIS,HID,HIE,HIP HE1 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(120.000);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HE1,bond,theta,phi,RATOM_CE1,RATOM_ND1,RATOM_CG);

    if (type==RESIDUE_HIE||type==RESIDUE_HIP){
        /* HIE,HIP HE2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(125.000);
        phi=degreetoradian(180.000);
        addAtom(RATOM_HE2,bond,theta,phi,RATOM_NE2,RATOM_CE1,RATOM_ND1);
    }

    /* HIS,HID,HIE,HIP HD2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(120.000);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HD2,bond,theta,phi,RATOM_CD2,RATOM_NE2,RATOM_CE1);

    break;

case RESIDUE_ILE:

    if (placement==RESIDUE_NTERMINUS) {
        /* ILE H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* ILE H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* ILE H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* ILE H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* ILE HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* ILE HB */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG1);

    /* ILE HG12 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HG12,bond,theta,phi,RATOM_CG1,RATOM_CB,RATOM_CD1);

    /* ILE HG13 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HG13,bond,theta,phi,RATOM_CG1,RATOM_CB,RATOM_CD1);

    /* ILE HD11 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    addAtom(RATOM_HD11,bond,theta,phi,RATOM_CD1,RATOM_CG1,RATOM_CB);

    /* ILE HD12 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HD12,bond,theta,phi,RATOM_CD1,RATOM_CG1,RATOM_CB);

    /* ILE HD13 */
```

```
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HD13,bond,theta,phi,RATOM_CD1,RATOM_CG1,RATOM_CB);

    /* ILE HG21 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    addAtom(RATOM_HG21,bond,theta,phi,RATOM_CG2,RATOM_CB,RATOM_CA);

    /* ILE HG22 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HG22,bond,theta,phi,RATOM_CG2,RATOM_CB,RATOM_CA);

    /* ILE HG23 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HG23,bond,theta,phi,RATOM_CG2,RATOM_CB,RATOM_CA);

    break;

case RESIDUE_LEU:

    if (placement==RESIDUE_NTERMINUS) {
        /* LEU H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* LEU H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* LEU H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* LEU H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* LEU HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* LEU HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* LEU HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* LEU HG */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HG,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CD1);

    /* LEU HD11 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    addAtom(RATOM_HD11,bond,theta,phi,RATOM_CD1,RATOM_CG,RATOM_CB);

    /* LEU HD12 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HD12,bond,theta,phi,RATOM_CD1,RATOM_CG,RATOM_CB);

    /* LEU HD13 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HD13,bond,theta,phi,RATOM_CD1,RATOM_CG,RATOM_CB);

    /* LEU HD21 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    addAtom(RATOM_HD21,bond,theta,phi,RATOM_CD2,RATOM_CG,RATOM_CB);

    /* LEU HD22 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HD22,bond,theta,phi,RATOM_CD2,RATOM_CG,RATOM_CB);

    /* LEU HD23 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HD23,bond,theta,phi,RATOM_CD2,RATOM_CG,RATOM_CB);

    break;

case RESIDUE_LYS:
case RESIDUE_LYN:

    if (placement==RESIDUE_NTERMINUS) {
        /* LYS H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* LYS H2 */
        /* bond: amber */
```

```
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* LYS H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* LYS,LYN H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* LYS,LYN HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* LYS,LYN HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* LYS,LYN HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* LYS,LYN HG2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HG2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CD);

    /* LYS,LYN HG3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HG3,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CD);

    /* LYS,LYN HD2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HD2,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_CE);

    /* LYS,LYN HD3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HD3,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_CE);

    /* LYS,LYN HE2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HE2,bond,theta,phi,RATOM_CE,RATOM_CD,RATOM_NZ);

    /* LYS,LYN HE3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HE3,bond,theta,phi,RATOM_CE,RATOM_CD,RATOM_NZ);

    if (type==RESIDUE_LYS) {
        /* LYS HZ1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.470);
        phi=degreetoradian(60.000);
        addAtom(RATOM_HZ1,bond,theta,phi,RATOM_NZ,RATOM_CE,RATOM_CD);
    }

    /* LYS,LYN HZ2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.470);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HZ2,bond,theta,phi,RATOM_NZ,RATOM_CE,RATOM_CD);

    /* LYS,LYN HZ3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.470);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HZ3,bond,theta,phi,RATOM_NZ,RATOM_CE,RATOM_CD);

    break;

case RESIDUE_MET:

    if (placement==RESIDUE_NTERMINUS) {
        /* MET H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* MET H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* MET H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* MET H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
```

```
            */
    }

    /* MET HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* MET HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* MET HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* MET HG2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HG2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_SD);

    /* MET HG3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HG3,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_SD);

    /* MET HE1 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    addAtom(RATOM_HE1,bond,theta,phi,RATOM_CE,RATOM_SD,RATOM_CG);

    /* MET HE2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HE2,bond,theta,phi,RATOM_CE,RATOM_SD,RATOM_CG);

    /* MET HE3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HE3,bond,theta,phi,RATOM_CE,RATOM_SD,RATOM_CG);

    break;

case RESIDUE_PHE:

    if (placement==RESIDUE_NTERMINUS) {
        /* PHE H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* PHE H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* PHE H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* PHE H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* PHE HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* PHE HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* PHE HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* PHE HD1 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(120.000);
    phi=degreetoradian(0.000);
    addAtom(RATOM_HD1,bond,theta,phi,RATOM_CD1,RATOM_CG,RATOM_CB);

    /* PHE HD2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(120.000);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HD2,bond,theta,phi,RATOM_CD2,RATOM_CE2,RATOM_CZ);

    /* PHE HE1 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(120.000);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HE1,bond,theta,phi,RATOM_CE1,RATOM_CD1,RATOM_CG);

    /* PHE HE2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(120.000);
    phi=degreetoradian(180.000);
```

```
    addAtom(RATOM_HE2,bond,theta,phi,RATOM_CE2,RATOM_CZ,RATOM_CE1);

    /* PHE HZ */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(120.000);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HZ,bond,theta,phi,RATOM_CZ,RATOM_CE1,RATOM_CD1);

    break;

case RESIDUE_PRO:


    if (placement==RESIDUE_NTERMINUS){
        /* PRO H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.5);
        phi=degreetoradian(120.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CD);

        /* PRO H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.5);
        phi=degreetoradian(240.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CD);

    }

    /* PRO HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(256.300);
    phi=degreetoradian(240);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CG,RATOM_CA);

    /* PRO HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CG,RATOM_CA);

    /* PRO HG2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HG2,bond,theta,phi,RATOM_CG,RATOM_CD,RATOM_CB);

    /* PRO HG3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HG3,bond,theta,phi,RATOM_CG,RATOM_CD,RATOM_CB);

    /* PRO HD2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240);
    addAtom(RATOM_HD2,bond,theta,phi,RATOM_CD,RATOM_N,RATOM_CG);

    /* PRO HD3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120);
    addAtom(RATOM_HD3,bond,theta,phi,RATOM_CD,RATOM_N,RATOM_CG);

    /* PRO HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    break;

case RESIDUE_SER:

    if (placement==RESIDUE_NTERMINUS) {
        /* SER H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* SER H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* SER H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

     }else{
        /* SER H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* SER HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* SER HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_OG);

    /* SER HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_OG);

    /* SER HG */
    /* bond: amber */
    /* angle: amber */
    bond=0.960;
    theta=degreetoradian(109.470);
    phi=degreetoradian(180.000);
```

```
    addAtom(RATOM_HG,bond,theta,phi,RATOM_OG,RATOM_CB,RATOM_CA);

    break;

case RESIDUE_THR:

    if (placement==RESIDUE_NTERMINUS) {
        /* THR H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* THR H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* THR H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* THR H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* THR HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* THR HB */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG2);

    /* THR HG1 */
    /* bond: amber */
    /* angle: amber */
    bond=0.960;
    theta=degreetoradian(109.470);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HG1,bond,theta,phi,RATOM_OG1,RATOM_CB,RATOM_CA);

    /* THR HG21 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    addAtom(RATOM_HG21,bond,theta,phi,RATOM_CG2,RATOM_CB,RATOM_CA);

    /* THR HG22 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(180.000);
    addAtom(RATOM_HG22,bond,theta,phi,RATOM_CG2,RATOM_CB,RATOM_CA);

    /* THR HG23 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HG23,bond,theta,phi,RATOM_CG2,RATOM_CB,RATOM_CA);

    break;

case RESIDUE_TRP:

    if (placement==RESIDUE_NTERMINUS) {
        /* TRP H1 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(60.000);
        addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* TRP H2 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

        /* TRP H3 */
        /* bond: amber */
        /* angle: amber */
        bond=1.010;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    }else{
        /* TRP H */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the HN depends on the phi angle,
            as its in plane with the previous C.
            modified for CA-CB reference phi;360-119.800 */
        bond=1.010;
        theta=degreetoradian(240.2);
        phi=degreetoradian(bb_phi-120);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
        /* Original entry;
            theta=degreetoradian(119.800);
            phi=degreetoradian(0.000);
            addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
            */
    }

    /* TRP HA */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
    /* Original entry;
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
        */

    /* TRP HB2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(120.000);
    addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

    /* TRP HB3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(240.000);
    addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);
```

```
/* TRP HD1 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(120.000);
phi=degreetoradian(0.000);
addAtom(RATOM_HD1,bond,theta,phi,RATOM_CD1,RATOM_CG,RATOM_CB);

/* TRP HE1 */
/* bond: amber */
/* angle: amber */
bond=1.010;
theta=degreetoradian(125.500);
phi=degreetoradian(180.000);
addAtom(RATOM_HE1,bond,theta,phi,RATOM_NE1,RATOM_CD1,RATOM_CG);

/* TRP HE3 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(120.000);
phi=degreetoradian(180.000);
addAtom(RATOM_HE3,bond,theta,phi,RATOM_CE3,RATOM_CZ3,RATOM_CH2);

/* TRP HZ2 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(120.000);
phi=degreetoradian(0.000);
addAtom(RATOM_HZ2,bond,theta,phi,RATOM_CZ2,RATOM_CE2,RATOM_NE1);

/* TRP HZ3 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(120.000);
phi=degreetoradian(180.000);
addAtom(RATOM_HZ3,bond,theta,phi,RATOM_CZ3,RATOM_CH2,RATOM_CZ2);

/* TRP HH2 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(120.000);
phi=degreetoradian(180.000);
addAtom(RATOM_HH2,bond,theta,phi,RATOM_CH2,RATOM_CZ2,RATOM_CE2);

break;

case RESIDUE_TYR:

if (placement==RESIDUE_NTERMINUS) {
    /* TYR H1 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    /* TYR H2 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.500);
    phi=degreetoradian(180.000);
    addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    /* TYR H3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

}else{
    /* TYR H */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the HN depends on the phi angle,
        as its in plane with the previous C.
        modified for CA-CB reference phi;360-119.800 */
    bond=1.010;
    theta=degreetoradian(240.2);
    phi=degreetoradian(bb_phi-120);
    addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
    /* Original entry;
        theta=degreetoradian(119.800);
        phi=degreetoradian(0.000);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
        */
}

/* TYR HA */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(120.000);
addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
/* Original entry;
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
    */

/* TYR HB2 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(120.000);
addAtom(RATOM_HB2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

/* TYR HB3 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(240.000);
addAtom(RATOM_HB3,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_CG);

/* TYR HD1 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(120.000);
phi=degreetoradian(0.000);
addAtom(RATOM_HD1,bond,theta,phi,RATOM_CD1,RATOM_CG,RATOM_CB);

/* TYR HD2 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(120.000);
phi=degreetoradian(180.000);
addAtom(RATOM_HD2,bond,theta,phi,RATOM_CD2,RATOM_CE2,RATOM_CZ);

/* TYR HE1 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(120.000);
phi=degreetoradian(180.000);
addAtom(RATOM_HE1,bond,theta,phi,RATOM_CE1,RATOM_CD1,RATOM_CG);

/* TYR HE2 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(120.000);
phi=degreetoradian(180.000);
addAtom(RATOM_HE2,bond,theta,phi,RATOM_CE2,RATOM_CZ,RATOM_CE1);

/* TYR HH */
/* bond: amber */
/* angle: amber */
bond=0.960;
theta=degreetoradian(113.000);
phi=degreetoradian(0.000);
addAtom(RATOM_HH,bond,theta,phi,RATOM_OH,RATOM_CZ,RATOM_CE1);

break;

case RESIDUE_VAL:

if (placement==RESIDUE_NTERMINUS) {
    /* VAL H1 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.500);
    phi=degreetoradian(60.000);
    addAtom(RATOM_H1,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    /* VAL H2 */
    /* bond: amber */
```

```
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.500);
    phi=degreetoradian(180.000);
    addAtom(RATOM_H2,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

    /* VAL H3 */
    /* bond: amber */
    /* angle: amber */
    bond=1.010;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_H3,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_C);

}else{
    /* VAL H */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the HN depends on the phi angle,
        as its in plane with the previous C.
        modified for CA-CB reference phi;360-119.800 */
    bond=1.010;
    theta=degreetoradian(240.2);
    phi=degreetoradian(bb_phi-120);
    addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_CA,RATOM_CB);
    /* Original entry;
        theta=degreetoradian(119.800);
        phi=degreetoradian(0.000);
        addAtom(RATOM_H,bond,theta,phi,RATOM_N,RATOM_RE_C,RATOM_PR_CA);
        */
}

/* VAL HA */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(120.000);
addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);
/* Original entry;
    bond=1.090;
    theta=degreetoradian(109.500);
    phi=degreetoradian(300.000);
    addAtom(RATOM_HA,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_RE_C);
    */

/* VAL HB */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(300.000);
addAtom(RATOM_HB,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

/* VAL HG11 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(60.000);
addAtom(RATOM_HG11,bond,theta,phi,RATOM_CG1,RATOM_CB,RATOM_CA);

/* VAL HG12 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(180.000);
addAtom(RATOM_HG12,bond,theta,phi,RATOM_CG1,RATOM_CB,RATOM_CA);

/* VAL HG13 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(300.000);
addAtom(RATOM_HG13,bond,theta,phi,RATOM_CG1,RATOM_CB,RATOM_CA);

/* VAL HG21 */
/* bond: amber */
/* angle: amber */
bond=1.090;
theta=degreetoradian(109.500);
phi=degreetoradian(60.000);
addAtom(RATOM_HG21,bond,theta,phi,RATOM_CG2,RATOM_CB,RATOM_CA);

/* VAL HG22 */
/* bond: amber */
/* angle: amber */
bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(180.000);
        addAtom(RATOM_HG22,bond,theta,phi,RATOM_CG2,RATOM_CB,RATOM_CA);

        /* VAL HG23 */
        /* bond: amber */
        /* angle: amber */
        bond=1.090;
        theta=degreetoradian(109.500);
        phi=degreetoradian(300.000);
        addAtom(RATOM_HG23,bond,theta,phi,RATOM_CG2,RATOM_CB,RATOM_CA);

        break;
    default:
        throw "Unable to build residue type";
    }
}

// build residue using backbone positions and chi angles
void Residue::buildchi(const double bb_phi,
                       const double bb_psi,
                       const float dchi[4]) {

    double bond;
    double theta;
    double phi;
    double chi[4];
    Vector3D v;
#ifdef CHECK
    int i;
    int cadef,cdef,ndef;
    static char error[256];
    ResidueInfo *ResidueInfoDataPtr;

    switch (placement) {
     case RESIDUE_CTERMINUS:
        ResidueInfoDataPtr=ResidueInfoDatact;
        break;
     case RESIDUE_CENTER:
        ResidueInfoDataPtr=ResidueInfoData;
        break;
     case RESIDUE_NTERMINUS:
        ResidueInfoDataPtr=ResidueInfoDatant;
        break;
    }

    cadef=cdef=ndef=0;
    for (i=0;i<RESIDUE_MAXATOM;i++) {
        if (atom[i].type==RATOM_C) {cdef++;continue;}
        if (atom[i].type==RATOM_CA) {cadef++;continue;}
        if (atom[i].type==RATOM_N) {ndef++;continue;}
        atom[i].type=RATOM_MAX;
    }
    if (cadef≠1 || cdef≠1 || ndef≠1)
      throw "Residue w/o predefined C,CA,N cannot be build";
#endif

    chi[0]=degreetoradian(dchi[0]);
    chi[1]=degreetoradian(dchi[1]);
    chi[2]=degreetoradian(dchi[2]);
    chi[3]=degreetoradian(dchi[3]);

    // Note : In this generation, CA,N,C is assumed to be fixed, and all
    // gerenation is done in reference to these atoms.

    switch (type) {

     case RESIDUE_MAX:
        // say what?
        throw "Illegal residue type (build)";
        break;

        // rebuild residue from chi angles.

     case RESIDUE_ALA:

        /* Atom ALA CB CH3E */
        /* bond: CH1E-CH3E */
        /* angle: NH1-CH1E-CH3E */
        bond=1.521;
        theta=degreetoradian(110.4);
        phi=-PI2/3.0;
        addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

        /* ALA O */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the O depends on the psi angle,
            as its in plane with the next N. */
```

```
bond=1.229;
theta=degreetoradian(120.500);
phi=degreetoradian(bb_psi);
addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
/* Original entry;
   theta=degreetoradian(120.500);
   phi=degreetoradian(0.000);
   addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
   */

if (placement==RESIDUE_CTERMINUS) {
    /* ALA OXT */
    /* bond: amber */
    /* angle: amber */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi+180.000);
    addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
}

break;

case RESIDUE_ARG:

/* Atom ARG CB CH2E */
/* bond: CH1E-CH2E */
/* theta: NH1-CH1E-CH2E */
bond=1.530;
theta=degreetoradian(110.5);
phi=-PI2/3.0;
addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

/* Atom ARG CG CH2E */
/* bond: CH2E-CH2E */
/* theta: CH1E-CH2E-CH2E */
/* AMBER: 1.525,109.470,180 */
bond=1.520;
theta=degreetoradian(114.1);
phi=chi[0];
addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

/* Atom ARG CD CH2E */
/* bond: CH2E-CH2E */
/* theta: CH2E-CH2E-CH2E */
/* AMBER: 1.525,109.470,180 */
bond=1.520;
theta=degreetoradian(111.3);
phi=chi[1];
addAtom(RATOM_CD,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

/* Atom ARG NE NH1 */
/* bond: CH2E-NH1 */
/* theta:CH2E-CH2E-NH1 */
/* AMBER: 1.480,111.000,180.0 */
bond=1.461;
theta=degreetoradian(112.0);
phi=chi[2];
addAtom(RATOM_NE,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_CB);

/* Atom ARG CZ C */
/* bond: NH1-C */
/* theta: CH2E-NH1-C */
/* AMBER: 1.330,123.000,180.0 */
bond=1.329;
theta=degreetoradian(124.2);
phi=chi[3];
addAtom(RATOM_CZ,bond,theta,phi,RATOM_NE,RATOM_CD,RATOM_CG);

/* Atom ARG NH1 NC2 */
/* bond: C-NC2 */
/* theta: NH1-C-NC2 */
/* AMBER: 1.330,122.000,0.0 */
bond=1.326;
theta=degreetoradian(120.0);
phi=degreetoradian(180.0);
addAtom(RATOM_NH1,bond,theta,phi,RATOM_CZ,RATOM_NE,RATOM_CD);

/* Atom ARG NH2 NH2 */
/* bond: C-NC2 */
/* theta: NH1-C-NC2 */
/* AMBER: 1.330,118.000,0.0 */
bond=1.326;
theta=degreetoradian(120.0);
phi=degreetoradian(0.0);
addAtom(RATOM_NH2,bond,theta,phi,RATOM_CZ,RATOM_NE,RATOM_CD);

/* ARG O */
/* bond: amber */
/* angle: amber */
/* The placement of the O depends on the psi angle,
   as its in plane with the next N. */
bond=1.229;
theta=degreetoradian(120.500);
phi=degreetoradian(bb_psi);
addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
/* Original entry;
   theta=degreetoradian(120.500);
   phi=degreetoradian(0.000);
   addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
  */

if (placement==RESIDUE_CTERMINUS){
   /* ARG OXT */
   /* bond: amber */
   /* angle: amber */
   bond=1.229;
   theta=degreetoradian(120.500);
   phi=degreetoradian(bb_psi+180.000);
   addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
}

break;

case RESIDUE_ASN:

/* Atom ASN CB CH2E */
/* bond: CH1E-CH2E */
/* theta: NH1-CH1E-CH2E */
bond=1.530;
theta=degreetoradian(110.5);
phi=-PI2/3.0;
addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

/* Atom ASN CG C */
/* bond: CH2E-C */
/* theta: CH1E-CH2E-C */
/* AMBER: 1.522,111.100,180.000 */
bond=1.516;
theta=degreetoradian(112.6);
phi=chi[0];
addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

/* Atom ASN OD1 O */
/* bond: C-O */
/* theta: CH2E-C-O */
/* AMBER: 1.229,120.500,0 */
bond=1.231;
theta=degreetoradian(120.8);
phi=chi[1];
addAtom(RATOM_OD1,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

/* Atom ASN ND2 NH2 */
/* bond: C-NH2 */
/* theta: CH2E-C-NH2 */
/* AMBER: 1.335,116.600,180 */
bond=1.328;
theta=degreetoradian(116.4);
phi=chi[1]-PI;
addAtom(RATOM_ND2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

/* ASN O */
/* bond: amber */
/* angle: amber */
/* The placement of the O depends on the psi angle,
   as its in plane with the next N. */
bond=1.229;
theta=degreetoradian(120.500);
phi=degreetoradian(bb_psi);
addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
/* Original entry;
   theta=degreetoradian(120.500);
   phi=degreetoradian(0.000);
   addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
  */

if (placement==RESIDUE_CTERMINUS){
   /* ASN OXT */
   /* bond: amber */
   /* angle: amber */
   bond=1.229;
   theta=degreetoradian(120.500);
   phi=degreetoradian(bb_psi+180.000);
   addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
}

break;

case RESIDUE_ASP:
case RESIDUE_ASH:
```

```
    /* Atom ASP,ASH CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom ASP,ASH CG C */
    /* bond: CH2E-C */
    /* theta: CH1E-CH2E-C */
    bond=1.516;
    theta=degreetoradian(112.6);
    phi=chi[0];
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom ASP,ASH OD1 OC */
    /* bond: C-OC */
    /* theta: CH2E-C-OC */
    bond=1.249;
    theta=degreetoradian(118.4);
    phi=chi[1];
    addAtom(RATOM_OD1,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom ASP,ASH OD2 OC */
    /* bond: C-OC */
    /* theta: CH2E-C-OC */
    bond=1.249;
    theta=degreetoradian(118.4);
    phi=chi[1]-PI;
    addAtom(RATOM_OD2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* ASP,ASH O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
        as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
        theta=degreetoradian(120.500);
        phi=degreetoradian(0.000);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
      */

    if (placement==RESIDUE_CTERMINUS){
        /* ASP OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

case RESIDUE_CYS:
case RESIDUE_CYM:
case RESIDUE_CYX:

    /* Atom CYS,CYM,CYX CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom CYS,CYM,CYX SG SG1E */
    /* bond: CH2E-SG1E */
    /* theta: CH1E-CH2E-SG1E */
    if (type==RESIDUE_CYX)
      bond=1.810;
    else
      bond=1.808;
    theta=degreetoradian(114.4);
    phi=chi[0];
    addAtom(RATOM_SG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* CYS,CYM,CYX O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
        as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
        theta=degreetoradian(120.500);
        phi=degreetoradian(0.000);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
      */

    if (placement==RESIDUE_CTERMINUS){
        /* CYS OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

case RESIDUE_GLN:

    /* Atom GLN CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom GLN CG CH2E */
    /* bond: CH2E-CH2E */
    /* theta: CH1E-CH2E-CH2E */
    /* AMBER: 1.525,109.470,180 */
    bond=1.520;
    theta=degreetoradian(114.1);
    phi=chi[0];
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom GLN CD C */
    /* bond: CH2E-C */
    /* theta: CH2E-CH2E-C */
    /* AMBER: 1.522,111.100,180.000 */
    bond=1.516;
    theta=degreetoradian(112.6);
    phi=chi[1];
    addAtom(RATOM_CD,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom GLN OE1 O */
    /* bond: C-O */
    /* theta: CH2E-C-O */
    /* AMBER: 1.229,120.500,0.000 */
    bond=1.231;
    theta=degreetoradian(120.8);
    phi=chi[2];
    addAtom(RATOM_OE1,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_CB);

    /* Atom GLN NE2 NH2 */
    /* bond: C-NH2 */
    /* theta: CH2E-C-NH2 */
    /* AMBER: 1.335,116.600,180.000 */
    bond=1.328;
    theta=degreetoradian(116.4);
    phi=chi[2]+PI;
    addAtom(RATOM_NE2,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_CB);

    /* GLN O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
        as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
        theta=degreetoradian(120.500);
        phi=degreetoradian(0.000);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
      */

    if (placement==RESIDUE_CTERMINUS){
        /* GLN OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }
```

```
    break;

case RESIDUE_GLU:
case RESIDUE_GLH:

    /* Atom GLU,GLH CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom GLU,GLH CG CH2E */
    /* bond: CH2E-CH2E */
    /* theta: CH1E-CH2E-CH2E */
    bond=1.520;
    theta=degreetoradian(114.1);
    phi=chi[0];
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom GLU,GLH CD C */
    /* bond: CH2E-C */
    /* theta: CH2E-CH2E-C */
    bond=1.516;
    theta=degreetoradian(112.6);
    phi=chi[1];
    addAtom(RATOM_CD,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom GLU,GLH OE1 OC */
    /* bond: C-OC */
    /* theta: CH2E-C-OC */
    bond=1.249;
    theta=degreetoradian(118.4);
    phi=chi[2];
    addAtom(RATOM_OE1,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_CB);

    /* Atom GLU,GLH OE2 OC */
    /* bond: C-OC */
    /* theta: CH2E-C-OC */
    bond=1.249;
    theta=degreetoradian(118.4);
    phi=chi[2]+PI;
    addAtom(RATOM_OE2,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_CB);

    /* GLU O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
        as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
        theta=degreetoradian(120.500);
        phi=degreetoradian(0.000);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
      */

    if (placement==RESIDUE_CTERMINUS){
        /* GLU OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

case RESIDUE_GLY:
    /* only a simple hydrogen */

    /* GLY O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
        as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
        theta=degreetoradian(120.500);
        phi=degreetoradian(0.000);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
      */

    if (placement==RESIDUE_CTERMINUS) {
        /* GLY OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

case RESIDUE_HIS:
case RESIDUE_HID:
case RESIDUE_HIE:
case RESIDUE_HIP:

    /* Atom HIS CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom HIS,HID,HIE,HIP CG C5 */
    /* bond: CH2E-C5 */
    /* theta: CH1E-CH2E-C5 */
    /* AMBER: 1.510,115.000,180.0 */
    bond=1.497;
    theta=degreetoradian(113.8);
    phi=theta;
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom HIS,HID,HIE,HIP ND1 NH1 */
    /* bond: C5-NH1 */
    /* theta: CH2E-C5-NH1 */
    /* AMBER: 1.390,122.0,180.0 */
    bond=1.378;
    theta=degreetoradian(122.7);
    phi=chi[1];
    addAtom(RATOM_ND1,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom HIS,HID,HIE,HIP CE1 CRH */
    /* bond: NH1-CRH */
    /* theta: C5-NH1-CRH */
    bond=1.345;
    theta=degreetoradian(109.0);
    phi=degreetoradian(180.0);
    addAtom(RATOM_CE1,bond,theta,phi,RATOM_ND1,RATOM_CG,RATOM_CB);

    /* Atom HIS,HID,HIE NE2 NR */
    /* bond: CRH-NR */
    /* theta: NH1-CRH-NR */
    bond=1.319;
    theta=degreetoradian(111.7);
    phi=degreetoradian(0.0);
    addAtom(RATOM_NE2,bond,theta,phi,RATOM_CE1,RATOM_ND1,RATOM_CG);

    /* Atom HIS,HID,HIE,HIP CD2 CR1E */
    /* bond: NR-CR1E */
    /* theta: CRH-NR-CR1E */
    bond=1.319;
    theta=degreetoradian(106.9);
    phi=degreetoradian(0.0);
    addAtom(RATOM_CD2,bond,theta,phi,RATOM_NE2,RATOM_CE1,RATOM_ND1);

    /* HIS,HID,HIE,HIP O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
        as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
        theta=degreetoradian(120.500);
        phi=degreetoradian(0.000);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
      */

    if (placement==RESIDUE_CTERMINUS){
        /* HID OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
```

```
    }

    break;

case RESIDUE_ILE:

    /* Atom ILE CB CH1E */
    /* bond: CH1E-CH1E */
    /* theta: NH1-CH1E-CH1E */
    bond=1.540;
    theta=degreetoradian(111.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom ILE CG1 CH2E */
    /* bond: CH1E-CH2E */
    /* theta: CH1E-CH1E-CH2E */
    /* AMBER: 1.525,109.470,180.000 */
    bond=1.530;
    theta=degreetoradian(110.4);
    phi=chi[0];
    addAtom(RATOM_CG1,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom ILE CD CH3E */
    /* bond: CH2E-CH3E */
    /* theta: CH1E-CH2E-CH3E */
    bond=1.513;
    theta=degreetoradian(113.8);
    phi=chi[1];
    addAtom(RATOM_CD1,bond,theta,phi,RATOM_CG1,RATOM_CB,RATOM_CA);

    /* Atom ILE CG2 CH3E */
    /* bond: CH1E-CH3E */
    /* theta: CH1E-CH1E-CH3E */
    /* AMBER: 1.525,109.470,60.000 */
    bond=1.521;
    theta=degreetoradian(110.5);
    phi=chi[0]-(PI2/3.0);
    addAtom(RATOM_CG2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* ILE O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
       as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
       theta=degreetoradian(120.500);
       phi=degreetoradian(0.000);
       addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
      */

    if (placement==RESIDUE_CTERMINUS) {
        /* ILE OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

case RESIDUE_LEU:

    /* Atom LEU CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom LEU CG CH1E */
    /* bond: CH2E-CH1E */
    /* theta: CH1E-CH2E-CH1E */
    /* AMBER: 1.525,109.470,180.000 */
    bond=1.530;
    theta=degreetoradian(116.3);
    phi=chi[0];
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom LEU CD1 CH3E */
    /* bond: CH1E-CH3E */
    /* theta: CH2E-CH1E-CH3E */
    /* AMBER: 1.525,109.470,180.000 */
    bond=1.521;
    theta=degreetoradian(110.7);
    phi=chi[1];
    addAtom(RATOM_CD1,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom LEU CD2 CH3E */
    /* bond: CH1E-CH3E */
    /* theta: CH2E-CH1E-CH3E */
    /* AMBER: 1.010,109.470,180.000 */
    bond=1.521;
    theta=degreetoradian(110.7);
    phi=chi[1]+PI2/3.0;
    addAtom(RATOM_CD2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* LEU O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
       as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
       theta=degreetoradian(120.500);
       phi=degreetoradian(0.000);
       addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
      */

    if (placement==RESIDUE_CTERMINUS) {
        /* LEU OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

case RESIDUE_LYS:
case RESIDUE_LYN:

    /* Atom LYS,LYN CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom LYS,LYN CG CH2E */
    /* bond: CH2E-CH2E */
    /* theta: CH1E-CH2E-CH2E */
    bond=1.520;
    theta=degreetoradian(114.1);
    phi=chi[0];
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom LYS,LYN CD CH2E */
    /* bond: CH2E-CH2E */
    /* theta: CH2E-CH2E-CH2E */
    bond=1.520;
    theta=degreetoradian(111.3);
    phi=chi[1];
    addAtom(RATOM_CD,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom LYS,LYN CE CH2E */
    /* bond: CH2E-CH2E */
    /* theta: CH2E-CH2E-CH2E */
    bond=1.520;
    theta=degreetoradian(111.3);
    phi=chi[2];
    addAtom(RATOM_CE,bond,theta,phi,RATOM_CD,RATOM_CG,RATOM_CB);

    /* Atom LYS,LYN NZ NH3 */
    /* bond: CH2E-NH3 */
    /* theta: CH2E-CH2E-NH3 */
    bond=1.489;
    theta=degreetoradian(111.9);
    phi=chi[3];
    addAtom(RATOM_NZ,bond,theta,phi,RATOM_CE,RATOM_CD,RATOM_CG);

    /* LYS,LYN O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
       as its in plane with the next N. */
    bond=1.229;
```

```
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
        theta=degreetoradian(120.500);
        phi=degreetoradian(0.000);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
       */

    if (placement==RESIDUE_CTERMINUS){
        /* LYS OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

case RESIDUE_MET:

    /* Atom MET CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom MET CG CH2E */
    /* bond: CH2E-CH2E */
    /* theta: CH1E-CH2E-CH2E */
    bond=1.520;
    theta=degreetoradian(114.1);
    phi=chi[0];
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom MET SD SM */
    /* bond: CH2E-SM */
    /* theta: CH2E-CH2E-SM */
    bond=1.803;
    theta=degreetoradian(112.7);
    phi=chi[1];
    addAtom(RATOM_SD,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom MET CE CH3E */
    /* bond: SM-CH3E */
    /* theta: CH2E-SM-CH3E */
    bond=1.791;
    theta=degreetoradian(100.9);
    phi=chi[2];
    addAtom(RATOM_CE,bond,theta,phi,RATOM_SD,RATOM_CG,RATOM_CB);

    /* MET O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
        as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
        theta=degreetoradian(120.500);
        phi=degreetoradian(0.000);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
       */

    if (placement==RESIDUE_CTERMINUS){
        /* MET OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

case RESIDUE_PHE:

    /* Atom PHE CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom PHE CG CF */
    /* bond: CH2E-CF */
    /* theta: CH1E-CH2E-CF */
    bond=1.502;
    theta=degreetoradian(113.8);
    phi=chi[0];
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom PHE CD1 CR1E */
    /* bond: CF-CR1E */
    /* theta: CH2E-CF-CR1E */
    bond=1.384;
    theta=degreetoradian(120.7);
    phi=chi[1];
    addAtom(RATOM_CD1,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom PHE CD2 CR1E */
    /* bond: CF-CR1E */
    /* theta: CH2E-CF-CR1E */
    bond=1.384;
    theta=degreetoradian(120.7);
    phi=chi[1]-PI;
    addAtom(RATOM_CD2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom PHE CE1 CR1E */
    /* bond: CR1E-CR1E */
    /* theta: CF-CR1E-CR1E */
    bond=1.382;
    theta=degreetoradian(120.7);
    phi=degreetoradian(180.0);
    addAtom(RATOM_CE1,bond,theta,phi,RATOM_CD1,RATOM_CG,RATOM_CB);

    /* Atom PHE CE2 CR1E */
    /* bond: CR1E-CR1E */
    /* theta: CF-CR1E-CR1E */
    bond=1.382;
    theta=degreetoradian(120.7);
    phi=degreetoradian(180.0);
    addAtom(RATOM_CE2,bond,theta,phi,RATOM_CD2,RATOM_CG,RATOM_CB);

    /* Atom PHE CZ CR1E */
    /* bond: CR1E-CR1E */
    /* theta: CR1E-CR1E-CR1E */
    bond=1.382;
    theta=degreetoradian(120.0);
    phi=degreetoradian(0.0);
    addAtom(RATOM_CZ,bond,theta,phi,RATOM_CE1,RATOM_CD1,RATOM_CG);

    /* PHE O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
        as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
        theta=degreetoradian(120.500);
        phi=degreetoradian(0.000);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
       */

    if (placement==RESIDUE_CTERMINUS){
        /* PHE OXT */
        /* bond: amber */
        /* angle: amber */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi+180.000);
        addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

case RESIDUE_PRO:

    /* Atom PRO CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: N-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(104.0);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom PRO CG CH2P */
    /* bond: CH2E-CH2P */
    /* theta: CH1E-CH2E-CH2P */
```

```
    bond=1.492;
    theta=degreetoradian(104.5);
    phi=chi[0];
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom PRO CD CH2P */
    /* bond: CH2P-CH2P */
    /* theta: CH1E-CH2P-CH2P */
    bond=1.503;
    theta=degreetoradian(106.1);
    phi=chi[1];
    addAtom(RATOM_CD,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* PRO O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
       as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    phi=degreetoradian(chi[1]);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
       theta=degreetoradian(120.500);
       phi=degreetoradian(0.000);
       addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
     */

    if (placement==RESIDUE_CTERMINUS){
       /* PRO OXT */
       /* bond: amber */
       /* angle: amber */
       bond=1.229;
       theta=degreetoradian(120.500);
       phi=degreetoradian(bb_psi+180.000);
       addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

  case RESIDUE_SER:

    /* Atom SER CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom SER OG OH1 */
    /* bond: CH2E-OH1 */
    /* theta: CH1E-CH2E-OH1 */
    /* AMBER: 1.430,109.470,180.000 */
    bond=1.417;
    theta=degreetoradian(111.1);
    phi=chi[0];
    addAtom(RATOM_OG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* SER O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
       as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
       theta=degreetoradian(120.500);
       phi=degreetoradian(0.000);
       addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
     */

    if (placement==RESIDUE_CTERMINUS) {
       /* SER OXT */
       /* bond: amber */
       /* angle: amber */
       bond=1.229;
       theta=degreetoradian(120.500);
       phi=degreetoradian(bb_psi+180.000);
       addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

  case RESIDUE_THR:

    /* Atom THR CB CH1E */
    /* bond: CH1E-CH1E */
    /* theta: NH1-CH1E-CH1E */
    bond=1.540;
    theta=degreetoradian(111.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom THR OG1 OH1 */
    /* bond: CH1E-OH1 */
    /* theta: CH1E-CH1E-OH1 */
    /* AMBER: 1.430,109.470,300.000 */
    bond=1.433;
    theta=degreetoradian(109.6);
    phi=chi[0];
    addAtom(RATOM_OG1,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom THR CG2 CH3E */
    /* bond: CH1E-CH3E */
    /* theta: CH1E-CH1E-CH3E */
    /* AMBER: 1.525,109.470,300.000 */
    bond=1.521;
    theta=degreetoradian(110.5);
    phi=chi[0]-(PI2/3.0);
    addAtom(RATOM_CG2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* THR O */
    /* bond: amber */
    /* angle: amber */
    /* The placement of the O depends on the psi angle,
       as its in plane with the next N. */
    bond=1.229;
    theta=degreetoradian(120.500);
    phi=degreetoradian(bb_psi);
    addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    /* Original entry;
       theta=degreetoradian(120.500);
       phi=degreetoradian(0.000);
       addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
     */

    if (placement==RESIDUE_CTERMINUS) {
       /* THR OXT */
       /* bond: amber */
       /* angle: amber */
       bond=1.229;
       theta=degreetoradian(120.500);
       phi=degreetoradian(bb_psi+180.000);
       addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
    }

    break;

  case RESIDUE_TRP:

    /* Atom TRP CB CH2E */
    /* bond: CH1E-CH2E */
    /* theta: NH1-CH1E-CH2E */
    bond=1.530;
    theta=degreetoradian(110.5);
    phi=-PI2/3.0;
    addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

    /* Atom TRP CG C5W */
    /* bond: CH2E-C5W */
    /* theta: CH1E-CH2E-C5W */
    bond=1.498;
    theta=degreetoradian(113.6);
    phi=chi[0];
    addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

    /* Atom TRP CD1 CR1E */
    /* bond: C5W-CR1E */
    /* theta: CH2E-C5W-CR1E */
    bond=1.365;
    theta=degreetoradian(126.9);
    phi=chi[1];
    addAtom(RATOM_CD1,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom TRP CD2 CW */
    /* bond: C5W-CW */
    /* theta: CH2E-C5W-CW */
    bond=1.433;
    theta=degreetoradian(126.8);
    phi=chi[1]-PI;
    addAtom(RATOM_CD2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

    /* Atom TRP NE1 NH1 */
    /* bond: CR1E-NH1 */
    /* theta: C5W-CR1E-NH1 */
    bond=1.374;
```

```
theta=degreetoradian(110.2);
phi=degreetoradian(180.0);
addAtom(RATOM_NE1,bond,theta,phi,RATOM_CD1,RATOM_CG,RATOM_CB);

/* Atom TRP CE2 CW */
/* bond: CW-CW */
/* theta: C5W-CW-CW */
bond=1.409;
theta=degreetoradian(107.2);
phi=degreetoradian(180.0);
addAtom(RATOM_CE2,bond,theta,phi,RATOM_CD2,RATOM_CG,RATOM_CB);

/* Atom TRP CE3 CR1E */
/* bond: CW-CR1E */
/* theta: CW-CW-CR1E */
bond=1.398;
theta=degreetoradian(118.8);
phi=degreetoradian(180.0);
addAtom(RATOM_CE3,bond,theta,phi,RATOM_CD2,RATOM_CE2,RATOM_NE1);

/* Atom TRP CZ2 CR1W */
/* bond: CW-CR1W */
/* theta: CW-CW-CR1W */
bond=1.394;
theta=degreetoradian(122.4);
phi=degreetoradian(0.0);
addAtom(RATOM_CZ2,bond,theta,phi,RATOM_CE2,RATOM_CD2,RATOM_CE3);

/* Atom TRP CZ3 CR1E */
/* bond: CR1E-CR1E */
/* theta: CW-CR1E-CR1E */
bond=1.382;
theta=degreetoradian(118.6);
phi=degreetoradian(0.0);
addAtom(RATOM_CZ3,bond,theta,phi,RATOM_CE3,RATOM_CD2,RATOM_CE2);

/* Atom TRP CH2 CR1W */
/* bond: CR1W-CR1W */
/* theta: CW-CR1W-CR1W */
bond=1.368;
theta=degreetoradian(117.5);
phi=degreetoradian(0.0);
addAtom(RATOM_CH2,bond,theta,phi,RATOM_CZ2,RATOM_CE2,RATOM_CD2);

/* TRP O */
/* bond: amber */
/* angle: amber */
/* The placement of the O depends on the psi angle,
   as its in plane with the next N. */
bond=1.229;
theta=degreetoradian(120.500);
phi=degreetoradian(bb_psi);
addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
/* Original entry;
   theta=degreetoradian(120.500);
   phi=degreetoradian(0.000);
   addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
  */

if (placement==RESIDUE_CTERMINUS){
   /* TRP OXT */
   /* bond: amber */
   /* angle: amber */
   bond=1.229;
   theta=degreetoradian(120.500);
   phi=degreetoradian(bb_psi+180.000);
   addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
}

break;

case RESIDUE_TYR:

/* Atom TYR CB CH2E */
/* bond: CH1E-CH2E */
/* theta: NH1-CH1E-CH2E */
bond=1.530;
theta=degreetoradian(110.5);
phi=-PI2/3.0;
addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

/* Atom TYR CG CY */
/* bond: CH2E-CY */
/* theta: CH1E-CH2E-CY */
bond=1.512;
theta=degreetoradian(113.9);
phi=chi[0];
addAtom(RATOM_CG,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

/* Atom TYR CD1 CR1E */
/* bond: CY-CR1E */
/* theta: CH2E-CY-CR1E */
bond=1.389;
theta=degreetoradian(120.8);
phi=chi[1];
addAtom(RATOM_CD1,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

/* Atom TYR CD2 CR1E */
/* bond: CY-CR1E */
/* theta:CH2E-CY-CR1E */
bond=1.389;
theta=degreetoradian(120.8);
phi=chi[1]-PI;
addAtom(RATOM_CD2,bond,theta,phi,RATOM_CG,RATOM_CB,RATOM_CA);

/* Atom TYR CE1 CR1E */
/* bond: CR1E-CR1E */
/* theta: CY-CR1E-CR1E */
bond=1.382;
theta=degreetoradian(121.2);
phi=degreetoradian(180.0);
addAtom(RATOM_CE1,bond,theta,phi,RATOM_CD1,RATOM_CG,RATOM_CB);

/* Atom TYR CE2 CR1E */
/* bond: CR1E-CR1E */
/* theta: CY-CR1E-CR1E */
bond=1.382;
theta=degreetoradian(121.2);
phi=degreetoradian(180.0);
addAtom(RATOM_CE2,bond,theta,phi,RATOM_CD2,RATOM_CG,RATOM_CB);

/* Atom TYR CZ CY2 */
/* bond: CR1E-CY2 */
/* theta: CR1E-CR1E-CY2 */
bond=1.378;
theta=degreetoradian(119.6);
phi=degreetoradian(0.0);
addAtom(RATOM_CZ,bond,theta,phi,RATOM_CE1,RATOM_CD1,RATOM_CG);

/* Atom TYR OH OH1 */
/* bond: CY2-OH1 */
/* theta: CR1E-CY2-OH1 */
bond=1.376;
theta=degreetoradian(119.9);
phi=degreetoradian(180.0);
addAtom(RATOM_OH,bond,theta,phi,RATOM_CZ,RATOM_CE2,RATOM_CD2);

/* TYR O */
/* bond: amber */
/* angle: amber */
/* The placement of the O depends on the psi angle,
   as its in plane with the next N. */
bond=1.229;
theta=degreetoradian(120.500);
phi=degreetoradian(bb_psi);
addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
/* Original entry;
   theta=degreetoradian(120.500);
   phi=degreetoradian(0.000);
   addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
  */

if (placement==RESIDUE_CTERMINUS){
   /* TYR OXT */
   /* bond: amber */
   /* angle: amber */
   bond=1.229;
   theta=degreetoradian(120.500);
   phi=degreetoradian(bb_psi+180.000);
   addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
}

break;

case RESIDUE_VAL:

/* Atom VAL CB CH1E */
/* bond: CH1E-CH1E */
/* theta: NH1-CH1E-CH1E */
bond=1.540;
theta=degreetoradian(111.5);
phi=-PI2/3.0;
addAtom(RATOM_CB,bond,theta,phi,RATOM_CA,RATOM_N,RATOM_C);

/* Atom VAL CG1 CH3E */
/* bond: CH1E-CH3E */
/* theta: CH1E-CH1E-CH3E */
/* AMBER: 1.525,109.470,60.00 */
bond=1.521;
theta=degreetoradian(110.5);
```

```
        phi=chi[0];
        addAtom(RATOM_CG1,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

        /* Atom VAL CG2 CH3E */
        /* bond: CH1E-CH3E */
        /* theta: CH1E-CH1E-CH3E */
        /* AMBER: 1.525,110.5,180.000 */
        bond=1.521;
        theta=degreetoradian(110.5);
        phi=chi[0]+(PI2/3.0);
        addAtom(RATOM_CG2,bond,theta,phi,RATOM_CB,RATOM_CA,RATOM_N);

        /* VAL O */
        /* bond: amber */
        /* angle: amber */
        /* The placement of the O depends on the psi angle,
           as its in plane with the next N. */
        bond=1.229;
        theta=degreetoradian(120.500);
        phi=degreetoradian(bb_psi);
        addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
        /* Original entry;
           theta=degreetoradian(120.500);
           phi=degreetoradian(0.000);
           addAtom(RATOM_O,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
          */

        if (placement==RESIDUE_CTERMINUS){
            /* VAL OXT */
            /* bond: amber */
            /* angle: amber */
            bond=1.229;
            theta=degreetoradian(120.500);
            phi=degreetoradian(bb_psi+180.000);
            addAtom(RATOM_OXT,bond,theta,phi,RATOM_C,RATOM_CA,RATOM_N);
        }

        break;
      default:
        throw "Unable to build residue type";
    }

    buildhydrogen(bb_phi,bb_psi);

#ifdef CHECK
    for (i=0;i<ResidueInfoDataPtr[type].atomcount;i++) {
        if (atom[i].type≠ResidueInfoDataPtr[type].atomtype[i]) {
            sprintf(error,"Unexpected result building "
                        "residue %s (atom %s is %s)",
                        ResidueInfoDataPtr[type].key,
                        AtomInfoData[ResidueInfoDataPtr[type].atomtype[i]].key,
                        AtomInfoData[atom[i].type].key);
            cerr≪error≪endl;
            throw error;
        }
    }
#endif

}
```

atoms.hh

```
#define RATOM_MAXTYPE 89
enum ResidueAtomtype {
RATOM_START=0,
RATOM_N=0,
RATOM_CA=1,
RATOM_C=2,
RATOM_O=3,
RATOM_CB=4,
RATOM_CD=5,
RATOM_CD1=6,
RATOM_CD2=7,
RATOM_CE=8,
RATOM_CE1=9,
RATOM_CE2=10,
RATOM_CE3=11,
RATOM_CG=12,
RATOM_CG1=13,
RATOM_CG2=14,
RATOM_CH2=15,
RATOM_CH3=16,
RATOM_CZ=17,
RATOM_CZ2=18,
RATOM_CZ3=19,
RATOM_NZ=20,
RATOM_OD1=21,
RATOM_OD2=22,
RATOM_OE1=23,
RATOM_OE2=24,
RATOM_OG=25,
RATOM_OG1=26,
RATOM_OH=27,
RATOM_OXT=28,
RATOM_SD=29,
RATOM_SG=30,
RATOM_H=31,
RATOM_H1=32,
RATOM_H2=33,
RATOM_H3=34,
RATOM_HA=35,
RATOM_HA2=36,
RATOM_HA3=37,
RATOM_HB=38,
RATOM_HB1=39,
RATOM_HB2=40,
RATOM_HB3=41,
RATOM_HD1=42,
RATOM_HD11=43,
RATOM_HD12=44,
RATOM_HD13=45,
RATOM_HD2=46,
RATOM_HD21=47,
RATOM_HD22=48,
RATOM_HD23=49,
RATOM_HD3=50,
RATOM_HE=51,
RATOM_HE1=52,
RATOM_HE2=53,
RATOM_HE21=54,
RATOM_HE22=55,
RATOM_HE3=56,
RATOM_HG=57,
RATOM_HG1=58,
RATOM_HG11=59,
RATOM_HG12=60,
RATOM_HG13=61,
RATOM_HG2=62,
RATOM_HG21=63,
RATOM_HG22=64,
RATOM_HG23=65,
RATOM_HG3=66,
RATOM_HH=67,
RATOM_HH11=68,
RATOM_HH12=69,
RATOM_HH2=70,
RATOM_HH21=71,
RATOM_HH22=72,
RATOM_HH31=73,
RATOM_HH32=74,
RATOM_HH33=75,
RATOM_HN=76,
RATOM_HSG=77,
RATOM_HZ=78,
RATOM_HZ1=79,
RATOM_HZ2=80,
RATOM_HZ3=81,
RATOM_ND1=82,
RATOM_ND2=83,
RATOM_NE=84,
RATOM_NE1=85,
RATOM_NE2=86,
RATOM_NH1=87,
RATOM_NH2=88,
RATOM_MAX};
```

atoms.cc

```
AtomInfo AtomInfoData[]={
"N", //0
"CA", //1
"C", //2
"O", //3
"CB", //4
"CD", //5
"CD1", //6
"CD2", //7
"CE", //8
"CE1", //9
"CE2", //10
"CE3", //11
"CG", //12
"CG1", //13
"CG2", //14
"CH2", //15
"CH3", //16
"CZ", //17
"CZ2", //18
"CZ3", //19
"NZ", //20
"OD1", //21
"OD2", //22
"OE1", //23
"OE2", //24
"OG", //25
"OG1", //26
"OH", //27
"OXT", //28
"SD", //29
"SG", //30
"H", //31
"H1", //32
"H2", //33
"H3", //34
"HA", //35
"HA2", //36
"HA3", //37
"HB", //38
"HB1", //39
"HB2", //40
"HB3", //41
"HD1", //42
"HD11", //43
"HD12", //44
"HD13", //45
"HD2", //46
"HD21", //47
"HD22", //48
"HD23", //49
"HD3", //50
"HE", //51
"HE1", //52
"HE2", //53
"HE21", //54
"HE22", //55
"HE3", //56
"HG", //57
"HG1", //58
"HG11", //59
"HG12", //60
"HG13", //61
"HG2", //62
"HG21", //63
"HG22", //64
"HG23", //65
"HG3", //66
"HH", //67
"HH11", //68
"HH12", //69
"HH2", //70
"HH21", //71
"HH22", //72
"HH31", //73
"HH32", //74
"HH33", //75
"HN", //76
"HSG", //77
"HZ", //78
"HZ1", //79
```

```
"HZ2", //80
"HZ3", //81
"ND1", //82
"ND2", //83
"NE", //84
"NE1", //85
"NE2", //86
"NH1", //87
"NH2", //88
"DU"};
```

residues.hh

```
#define RRESIDUE_MAXTYPE 29
enum Residuetype {
RESIDUE_START=0,
RESIDUE_ACE=0,
RESIDUE_ALA,
RESIDUE_ARG,
RESIDUE_ASH,
RESIDUE_ASN,
RESIDUE_ASP,
RESIDUE_CYM,
RESIDUE_CYS,
RESIDUE_CYX,
RESIDUE_GLH,
RESIDUE_GLN,
RESIDUE_GLU,
RESIDUE_GLY,
RESIDUE_HID,
RESIDUE_HIE,
RESIDUE_HIP,
RESIDUE_HIS,
RESIDUE_ILE,
RESIDUE_LEU,
RESIDUE_LYN,
RESIDUE_LYS,
RESIDUE_MET,
RESIDUE_PHE,
RESIDUE_PRO,
RESIDUE_SER,
RESIDUE_THR,
RESIDUE_TRP,
RESIDUE_TYR,
RESIDUE_VAL,
RESIDUE_MAX};
```

residues.cc

```
#define RESIDUEKEY_MAX 29
ResidueInfo ResidueInfoData[RESIDUEKEY_MAX] ={
{"Ace beginning group","ACE",6, //0
 {RATOM_C,RATOM_O,RATOM_CH3,RATOM_HH31,RATOM_HH32,RATOM_HH33,RATOM_MAX},
 {0.59720,-0.56790,-0.36620,0.11230,0.11230,0.11230,0},
 {1.9080,1.6612,1.9080,1.4870,1.4870,1.4870,0},
 {0.0860,0.2100,0.1094,0.0157,0.0157,0.0157,0},
 {-1,-1,0,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,2,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,3,4,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Alanine","ALA",10, //1
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_H,RATOM_HA,RATOM_HB1,RATOM_HB2,RATOM_HB3,RATOM_MAX},
 {-0.41570,0.03370,0.59730,-0.56790,-0.18250,0.27190,0.08230,0.06030,0.06030,0.06030,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,0.6000,1.3870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,6,-1,-1,-1,7,8,9,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Arginine","ARG",24, //2
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_CZ,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HE,RATOM_HG2,RATOM_HG3,RATOM_HH11,RATOM_HH12,RATOM_HH21,RATOM_HH22,RATOM_NE,RATOM_NH1,RATOM_NH2,RATOM_MAX},
 {-0.34790,-0.26370,0.73410,-0.58940,-
0.00070,0.04860,0.03900,0.80760,0.27470,0.15600,0.03270,0.03270,0.06870,0.06870,0.34560,0.02850,0.02850,0.44780,0.44780,0.44780,0.44780,-0.52950,-0.86270,-0.86270,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,1.3870,1.3870,0.6000,1.4870,1.4870,0.6000,0.6000,0.6000,0.6000,1.8240,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.1700,0.1700,0.1700,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,12,-1,-1,-1,13,14,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,15,-1,-1,-1,16,-
1,17,18,-1,19,20,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,21,-1,-1,22,23,-1}},
{"Asp neutral","ASH",13, //3
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG,RATOM_OD1,RATOM_OD2,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_MAX},
 {-0.41570,0.03410,0.59730,-0.56790,-0.03160,0.64620,-0.55540,-0.63760,0.27190,0.08640,0.04880,0.04880,0.47470,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.6612,1.7210,0.6000,1.3870,1.4870,1.4870,0.0000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.2100,0.2104,0.0157,0.0157,0.0157,0.0157,0.0000,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,6,7,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Asparagine","ASN",14, //4
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG,RATOM_OD1,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD21,RATOM_HD22,RATOM_ND2,RATOM_MAX},
 {-0.41570,0.01430,0.59730,-0.56790,-0.20410,0.71300,-0.59310,0.27190,0.10480,0.07970,0.07970,0.41960,0.41960,-0.91910,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,0.6000,0.6000,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.1700,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,-1,8,-1,-1,-1,-1,9,10,-1,-1,-1,-1,-1,11,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,13,-1,-1,-1,-1,-1,-1}},
{"Aspartic acid","ASP",12, //5
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG,RATOM_OD1,RATOM_OD2,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_MAX},
 {-0.51630,0.03810,0.53660,-0.58190,-0.03030,0.79940,-0.80140,-0.80140,0.29360,0.08800,-0.01220,-0.01220,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.6612,1.6612,0.6000,1.3870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.2100,0.2100,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,6,7,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Cysteine with negative charge","CYM",10, //6
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_SG,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HN,RATOM_MAX},
 {-0.41570,-0.03510,0.59730,-0.56790,-0.24130,-0.88440,0.05080,0.11220,0.11220,0.27190,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,2.0000,1.3870,1.3870,1.3870,0.6000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2500,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,6,-1,-1,-1,-1,7,8,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,9,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Cysteine","CYS",11, //7
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_SG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HG,RATOM_MAX},
 {-0.41570,0.02130,0.59730,-0.56790,-0.12310,-0.31190,0.27190,0.11240,0.11120,0.11120,0.19330,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,2.0000,0.6000,1.3870,1.3870,1.3870,0.6000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2500,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,6,-1,-1,-1,7,-1,-1,-1,-1,8,9,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Cystine(s-s bridge)","CYX",10, //8
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_SG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_MAX},
 {-0.41570,0.04290,0.59730,-0.56790,-0.07900,-0.10810,0.27190,0.07660,0.09100,0.09100,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,2.0000,0.6000,1.3870,1.3870,1.3870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2500,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,6,-1,-1,-1,7,-1,-1,-1,-1,8,9,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Glu neutral","GLH",16, //9
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_OE1,RATOM_OE2,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HE2,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {-0.41570,0.01450,0.59730,-0.56790,-0.00710,0.68010,-0.01740,-0.58380,-0.65110,0.27190,0.07790,0.02560,0.02560,0.46410,0.04300,0.04300,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,1.7210,0.6000,1.3870,1.4870,1.4870,0.0000,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.1094,0.2100,0.2104,0.0157,0.0157,0.0157,0.0157,0.0000,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,8,-1,-1,-1,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,13,-1,-1,-1,-1,-1,-1,-1,-1,14,-1,-1,-1,15,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Glutamine","GLN",17, //10
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_OE1,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HE21,RATOM_HE22,RATOM_HG2,RATOM_HG3,RATOM_NE2,RATOM_MAX},
 {-0.41570,-0.00310,0.59730,-0.56790,-0.00360,0.69510,-0.06450,-0.60860,0.27190,0.08500,0.01710,0.01710,0.42510,0.42510,0.03520,0.03520,-0.94070,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,0.6000,0.6000,1.4870,1.4870,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.1094,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.1700,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,12,13,-1,-1,-1,-1,-1,-1,14,-1,-1,-1,15,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,16,-1,-1,-1}},
{"Glutamic acid","GLU",15, //11
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_OE1,RATOM_OE2,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {-0.51630,0.03970,0.53660,-0.58190,0.05600,0.80540,0.01360,-0.81880,-0.81880,0.29360,0.11050,-0.01730,-0.01730,-0.04250,-0.04250,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,1.6612,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.1094,0.2100,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,8,-1,-1,-1,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,13,-1,-1,-1,14,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Glycine","GLY",7, //12
```

```
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_H,RATOM_HA2,RATOM_HA3,RATOM_MAX},
 {-0.41570,-0.02520,0.59730,-0.56790,0.27190,0.06980,0.06980,0},
 {1.8240,1.9080,1.9080,1.6612,0.6000,1.3870,1.3870,0},
 {0.1700,0.1094,0.0860,0.2100,0.0157,0.0157,0.0157,0},
 {0,1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,4,-1,-1,-1,5,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Histidine deltah","HID",17, //13
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {-0.41570,0.01880,0.59730,-0.56790,-0.04620,0.12920,0.20570,-0.02660,0.27190,0.08810,0.04020,0.04020,0.36490,0.11470,0.13920,-0.38110,-0.57270,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,0.6000,1.4090,1.3590,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,12,-1,-1,-1,13,-1,-1,-1,-1,-1,14,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,15,-1,-1,-1,16,-1,-1,-1}},
{"Histidine epsilonh","HIE",17, //14
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {-0.41570,-0.05810,0.59730,-0.56790,-0.00740,-0.22070,0.16350,0.18680,0.27190,0.13600,0.03670,0.03670,0.18620,0.14350,0.33390,-0.54320,-0.27950,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,1.4090,1.3590,0.6000,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0157,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,12,-1,-1,-1,-1,-1,13,14,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,15,-1,-1,-1,16,-1,-1,-1}},
{"Histidine plus","HIP",18, //15
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {-0.34790,-0.13540,0.73410,-0.58940,-0.04140,-0.11410,-0.01700,-0.00120,0.27470,0.12120,0.08100,0.08100,0.38660,0.23170,0.26810,0.39110,-0.15130,-0.17180,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,0.6000,1.4090,1.3590,0.6000,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0157,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,12,-1,-1,-1,13,-1,-1,-1,-1,-1,14,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,16,-1,-1,-1,17,-1,-1,-1}},
{"Histidine","HIS",17, //16
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {-0.41570,0.01880,0.59730,-0.56790,-0.04620,0.12920,0.20570,-0.02660,0.27190,0.08810,0.04020,0.04020,0.36490,0.11470,0.13920,-0.38110,-0.57270,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,0.6000,1.4090,1.3590,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,12,-1,-1,-1,13,-1,-1,-1,-1,-1,14,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,15,-1,-1,-1,16,-1,-1,-1}},
{"Isoleucine","ILE",19, //17
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CG1,RATOM_CG2,RATOM_H,RATOM_HA,RATOM_HB,RATOM_HD11,RATOM_HD12,RATOM_HD13,RATOM_HG12,RATOM_HG13,RATOM_HG21,RATOM_HG22,RATOM_HG23,RATOM_MAX},
 {-0.41570,-0.05970,0.59730,-0.56790,0.13030,-0.06600,-0.04300,-0.32040,0.27190,0.08690,0.01870,0.01860,0.01860,0.01860,0.02360,0.02360,0.08820,0.08820,0.08820,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,5,-1,-1,-1,-1,-1,-1,6,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,10,-1,-1,-1,-1,11,12,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,14,15,-1,16,17,18,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Leucine","LEU",19, //18
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD11,RATOM_HD12,RATOM_HD13,RATOM_HD21,RATOM_HD22,RATOM_HD23,RATOM_HG,RATOM_MAX},
 {-0.41570,-0.05180,0.59730,-0.56790,-0.11020,-0.41210,-0.41210,0.35310,0.27190,0.09220,0.04570,0.04570,0.10000,0.10000,0.10000,0.10000,0.10000,0.10000,-0.03610,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,5,6,-1,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,-1,12,13,14,-1,15,16,17,-1,-1,-1,-1,-1,-1,-1,-1,18,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Lysine neutral","LYN",21, //19
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CE,RATOM_CG,RATOM_NZ,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HE2,RATOM_HE3,RATOM_HG2,RATOM_HG3,RATOM_HZ2,RATOM_HZ3,RATOM_MAX},
 {-0.41570,-0.07206,0.59730,-0.56790,-0.04845,-0.03768,0.32604,0.06612,-1.03581,0.27190,0.09940,0.03400,0.03400,0.01155,0.01155,-0.03358,-
0.03358,0.01041,0.01041,0.38604,0.38604,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.8240,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,1.1000,1.1000,1.4870,1.4870,0.6000,0.6000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.1700,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,6,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,13,-1,-1,-1,14,-1,-1,15,-1,-1,16,-1,-1,-1,-1,-1,17,-1,-1,-1,18,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,19,20,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Lysine","LYS",22, //20
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CE,RATOM_CG,RATOM_NZ,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HE2,RATOM_HE3,RATOM_HG2,RATOM_HG3,RATOM_HZ1,RATOM_HZ2,RATOM_HZ3,RATOM_MAX},
 {-0.34790,-0.24000,0.73410,-0.58940,-0.00940,-0.04790,-0.01430,0.01870,-
0.38540,0.27470,0.14260,0.03620,0.03620,0.06210,0.06210,0.11350,0.11350,0.01030,0.01030,0.34000,0.34000,0.34000,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.8240,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,1.1000,1.1000,1.4870,1.4870,0.6000,0.6000,0.6000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.1700,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,6,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,13,-1,-1,-1,14,-1,-1,15,-1,-1,16,-1,-1,-1,-1,-1,17,-1,-1,-1,18,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,19,20,21,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Methionine","MET",17, //21
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CE,RATOM_CG,RATOM_SD,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HE1,RATOM_HE2,RATOM_HE3,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {-0.41570,-0.02370,0.59730,-0.56790,0.03420,-0.05360,0.00180,-0.27370,0.27190,0.08800,0.02410,0.02410,0.06840,0.06840,0.06840,0.04400,0.04400,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,2.0000,0.6000,1.3870,1.4870,1.4870,1.3870,1.3870,1.3870,1.3870,1.3870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.2500,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,5,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,12,13,-1,-1,14,-1,-1,-1,-1,-1,15,-1,-1,-1,16,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Phenylalanine","PHE",20, //22
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CE1,RATOM_CE2,RATOM_CG,RATOM_CZ,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_HZ,RATOM_MAX},
 {-0.41570,-0.00240,0.59730,-0.56790,-0.03430,-0.12560,-0.12560,-0.17040,-0.17040,0.01180,-0.10720,0.27190,0.09780,0.02950,0.02950,0.13300,0.13300,0.14300,0.14300,0.12970,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,1.4590,1.4590,1.4590,1.4590,1.4590,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0150,0.0150,0.0150,0},
 {0,1,2,3,4,-1,5,6,-1,7,8,-1,9,-1,-1,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,11,-1,-1,-1,12,-1,-1,-1,-1,13,14,15,-1,-1,-1,16,-1,-1,-1,-1,-1,17,18,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,19,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Proline","PRO",14, //23
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {-0.25480,-0.02660,0.58960,-0.57480,-0.00700,0.01920,0.01890,0.06410,0.02530,0.02530,0.03910,0.03910,0.02130,0.02130,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.3870,1.4870,1.4870,1.3870,1.3870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,-1,-1,8,9,-1,-1,-1,-1,10,-1,-1,-1,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,12,-1,-1,-1,13,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Serine","SER",11, //24
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_OG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HG,RATOM_MAX},
 {-0.41570,-0.02490,0.59730,-0.56790,0.21170,-0.65460,0.27190,0.08430,0.03520,0.03520,0.42750,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.7210,0.6000,1.3870,1.3870,1.3870,0.0000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2104,0.0157,0.0157,0.0157,0.0157,0.0000,0},
```

```
{0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,6,-1,-1,-1,7,-1,-1,-1,-1,8,9,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Threonine","THR",14, //25
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG2,RATOM_OG1,RATOM_H,RATOM_HA,RATOM_HB,RATOM_HG1,RATOM_HG21,RATOM_HG22,RATOM_HG23,RATOM_MAX},
 {-0.41570,-0.03890,0.59730,-0.56790,0.36540,-0.24380,-0.67610,0.27190,0.10070,0.00430,0.41020,0.06420,0.06420,0.06420,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.7210,0.6000,1.3870,1.3870,0.0000,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.2104,0.0157,0.0157,0.0157,0.0000,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,7,-1,-1,-1,8,-1,-1,9,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,10,-1,-1,-1,-1,11,12,13,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Tryptophan","TRP",24, //26
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CE2,RATOM_CE3,RATOM_CG,RATOM_CH2,RATOM_CZ2,RATOM_CZ3,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HE1,RATOM_HE3,RATOM_HH2,RATOM_HZ2,RATOM_HZ3,RATOM_NE1,RATOM_MAX},
 {-0.41570,-0.02750,0.59730,-0.56790,-0.00500,-0.16380,0.12430,0.13800,-0.23870,-0.14150,-0.11340,-0.26010,-
0.19720,0.27190,0.11230,0.03390,0.03390,0.20620,0.34120,0.17000,0.14170,0.15720,0.14470,-0.34180,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,1.4090,0.6000,1.4590,1.4590,1.4590,1.4590,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0150,0.0157,0.0150,0.0150,0.0150,0.0150,0.1700,0},
 {0,1,2,3,4,-1,5,6,-1,-1,7,8,9,-1,-1,10,-1,-1,11,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,13,-1,-1,-1,14,-1,-1,-1,-1,15,16,17,-1,-1,-1,-1,-1,-1,-1,-1,-1,18,-1,-1,-1,19,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,20,-1,-1,-1,-1,-1,-1,-1,-1,-1,21,22,-1,-1,-1,23,-1,-1,-1,-1}},
{"Tyrosine","TYR",21, //27
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CE1,RATOM_CE2,RATOM_CG,RATOM_CZ,RATOM_OH,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_HH,RATOM_MAX},
 {-0.41570,-0.00140,0.59730,-0.56790,-0.01520,-0.19060,-0.19060,-0.23410,-0.23410,-0.00110,0.32260,-
0.55790,0.27190,0.08760,0.02950,0.02950,0.16990,0.16990,0.16560,0.16560,0.39920,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.7210,0.6000,1.3870,1.4870,1.4870,1.4590,1.4590,1.4590,1.4590,0.0000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.2104,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0150,0.0150,0.0000,0},
 {0,1,2,3,4,-1,5,6,-1,7,8,-1,9,-1,-1,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,11,-1,-1,-1,12,-1,-1,-1,13,-1,-1,-1,-1,14,15,16,-1,-1,-1,17,-1,-1,-1,-1,-1,18,19,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,20,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Valine","VAL",16, //28
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG1,RATOM_CG2,RATOM_H,RATOM_HA,RATOM_HB,RATOM_HG11,RATOM_HG12,RATOM_HG13,RATOM_HG21,RATOM_HG22,RATOM_HG23,RATOM_MAX},
 {-0.41570,-0.08750,0.59730,-0.56790,0.29850,-0.31920,-0.31920,0.27190,0.09690,-0.02970,0.07910,0.07910,0.07910,0.07910,0.07910,0.07910,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,5,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,-1,8,-1,-1,9,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,10,11,12,-1,13,14,15,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}}
};
```

residuesct.cc

```
ResidueInfo ResidueInfoDatact[RESIDUEKEY_MAX] ={
{"Unavailable","ACE",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Alanine coo- anion","ALA",11, //1
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB1,RATOM_HB2,RATOM_HB3,RATOM_MAX},
 {-0.38210,-0.17470,0.77310,-0.80550,-0.20930,-0.80550,0.26810,0.10670,0.07640,0.07640,0.07640,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,6,-1,-1,-1,7,-1,-1,-1,8,9,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Arginine coo- anion","ARG",25, //2
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_CZ,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HE,RATOM_HG2,RATOM_HG3,RATOM_HH11,RATOM_HH12,RATOM_HH21,RATOM_HH22,RATOM_NE,RATOM_NH1,RATOM_NH2,RATOM_MAX},
 {-0.34810,-0.30680,0.85570,-0.82660,-0.03740,0.11140,0.07440,0.83680,-
0.82660,0.27640,0.14470,0.03710,0.03710,0.04680,0.04680,0.34790,0.01850,0.01850,0.44930,0.44930,0.44930,0.44930,-0.55640,-0.87370,-0.87370,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,1.3870,1.3870,0.6000,1.4870,1.4870,0.6000,0.6000,0.6000,0.6000,1.8240,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.1700,0.1700,0.1700,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,13,-1,-1,-1,14,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,16,-1,-1,-1,17,-
1,18,19,-1,20,21,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,22,-1,-1,23,24,-1}},
{"Unavailable","ASH",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Asparagine coo- anion","ASN",15, //4
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG,RATOM_OD1,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD21,RATOM_HD22,RATOM_ND2,RATOM_MAX},
 {-0.38210,-0.20800,0.80500,-0.81470,-0.22990,0.71530,-0.60100,-0.81470,0.26810,0.13580,0.10230,0.10230,0.41500,0.41500,-0.90840,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.6612,1.6612,0.6000,1.3870,1.4870,1.4870,0.6000,0.6000,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.2100,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.1700,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,7,-1,-1,8,-1,-1,-1,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,-1,12,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,14,-1,-1,-1,-1,-1,-1}},
{"Aspartic acid coo- anion","ASP",13, //5
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG,RATOM_OD1,RATOM_OD2,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_MAX},
 {-0.51920,-0.18170,0.72560,-0.78870,-0.06770,0.88510,-0.81620,-0.81620,-0.78870,0.30550,0.10460,-0.02120,-0.02120,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.6612,1.6612,1.6612,0.6000,1.3870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.2100,0.2100,0.2100,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,6,7,-1,-1,-1,-1,-1,8,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Unavailable","CYH",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Cysteine coo- anion","CYS",12, //7
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_OXT,RATOM_SG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HSG,RATOM_MAX},
 {-0.38210,-0.16350,0.74970,-0.79810,-0.19960,-0.79810,-0.31020,0.26810,0.13960,0.14370,0.14370,0.20680,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.6612,2.0000,0.6000,1.3870,1.3870,1.3870,0.6000,0},
```

```
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2100,0.2500,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,6,7,-1,-1,-1,8,-1,-1,-1,-1,9,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Cystine(s-s bridge) coo- anion","CYX",11, //8
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_OXT,RATOM_SG,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_MAX},
 {-0.38210,-0.13180,0.76180,-0.80410,-0.19430,-0.80410,-0.05290,0.26810,0.09380,0.12280,0.12280,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.6612,2.0000,0.6000,1.3870,1.3870,1.3870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2100,0.2500,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,6,7,-1,-1,-1,8,-1,-1,-1,-1,9,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Unavailable","GLH",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Glutamine coo- anion","GLN",18, //10
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_OE1,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HE21,RATOM_HE22,RATOM_HG2,RATOM_HG3,RATOM_NE2,RATOM_MAX},
 {-0.38210,-0.22480,0.77750,-0.80420,-0.06640,0.70930,-0.02100,-0.60980,-0.80420,0.26810,0.12320,0.04520,0.04520,0.43040,0.43040,0.02030,0.02030,-0.95740,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,1.6612,0.6000,1.3870,1.4870,1.4870,0.6000,0.6000,1.4870,1.4870,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.1094,0.2100,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.1700,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,-1,-1,8,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,13,14,-1,-1,-1,-1,-1,15,-1,-1,-1,16,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,17,-1,-1,-1}},
{"Glutamic acid coo- anion","GLU",16, //11
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_OE1,RATOM_OE2,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {-0.51920,-0.20590,0.74200,-0.79300,0.00710,0.81830,0.06750,-0.82200,-0.82200,-0.79300,0.30550,0.13990,-0.00780,-0.00780,-0.05480,-0.05480,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,1.6612,1.6612,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.1094,0.2100,0.2100,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,8,-1,-1,-1,9,-1,-1,10,-1,-1,-1,11,-1,-1,-1,-1,12,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,14,-1,-1,-1,15,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Glycine coo- anion","GLY",8, //12
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_OXT,RATOM_H,RATOM_HA2,RATOM_HA3,RATOM_MAX},
 {-0.38210,-0.24930,0.72310,-0.78550,-0.78550,0.26810,0.10560,0.10560,0},
 {1.8240,1.9080,1.9080,1.6612,1.6612,0.6000,1.3870,1.3870,0},
 {0.1700,0.1094,0.0860,0.2100,0.2100,0.0157,0.0157,0.0157,0},
 {0,1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,4,-1,-1,5,-1,-1,-1,-1,6,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Histidine deltah coo- anion","HID",18, //13
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {-0.38210,-0.17390,0.76150,-0.80160,-0.10460,0.10010,0.19250,0.02930,-0.80160,0.26810,0.11000,0.05650,0.05650,0.37550,0.12410,0.14180,-0.38920,-0.56290,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,0.6000,1.4090,1.3590,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,13,-1,-1,-1,14,-1,-1,-1,-1,-1,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,16,-1,-1,-1,17,-1,-1,-1}},
{"Histidine epsilon-h coo- anion","HIE",18, //14
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {-0.38210,-0.26990,0.79160,-0.80650,-0.10680,-0.25880,0.15580,0.27240,-0.80650,0.26810,0.16500,0.06200,0.06200,0.19570,0.14480,0.33190,-0.55170,-0.26700,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,1.4090,1.3590,0.6000,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0157,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,13,-1,-1,-1,-1,-1,14,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,16,-1,-1,-1,17,-1,-1,-1}},
{"Histidine plus coo-","HIP",19, //15
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {-0.34810,-0.14450,0.80320,-0.81770,-0.08000,-0.12560,-0.02510,0.02980,-0.81770,0.27640,0.11150,0.08680,0.08680,0.38830,0.23360,0.26940,0.39130,-0.15010,-0.16830,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,0.6000,1.4090,1.3590,0.6000,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0157,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,13,-1,-1,-1,14,-1,-1,-1,-1,-1,15,16,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,17,-1,-1,-1,18,-1,-1,-1}},
{"Histidine","HIS",18, //16
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {-0.38210,-0.17390,0.76150,-0.80160,-0.10460,0.10010,0.19250,0.02930,-0.80160,0.26810,0.11000,0.05650,0.05650,0.37550,0.12410,0.14180,-0.38920,-0.56290,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,0.6000,1.4090,1.3590,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,13,-1,-1,-1,14,-1,-1,-1,-1,-1,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,16,-1,-1,-1,17,-1,-1,-1}},
{"Isoleucine coo- anion","ILE",20, //17
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CG1,RATOM_CG2,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB,RATOM_HD11,RATOM_HD12,RATOM_HD13,RATOM_HG12,RATOM_HG13,RATOM_HG21,RATOM_HG22,RATOM_HG23,RATOM_MAX},
 {-0.38210,-0.31000,0.83430,-0.81900,0.03630,-0.06990,-0.03230,-0.34980,-0.81900,0.26810,0.13750,0.07660,0.01960,0.01960,0.01960,0.03210,0.03210,0.10210,0.10210,0.10210,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,5,-1,-1,-1,-1,-1,-1,6,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,9,-1,-1,-1,10,-1,-1,11,-1,-1,-1,-1,12,13,14,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,15,16,-1,17,18,19,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Leucine coo- anion","LEU",20, //18
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CG,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD11,RATOM_HD12,RATOM_HD13,RATOM_HD21,RATOM_HD22,RATOM_HD23,RATOM_HG,RATOM_MAX},
 {-0.38210,-0.28470,0.83260,-0.81990,-0.24690,-0.41630,-0.41630,0.37060,-0.81990,0.26810,0.13460,0.09740,0.09740,0.10380,0.10380,0.10380,0.10380,0.10380,0.10380,-0.03740,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,5,6,-1,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,13,14,15,-1,16,17,18,-1,-1,-1,-1,-1,-1,-1,19,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Unavailable","LYN",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Lysine coo- anion","LYS",23, //20
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CE,RATOM_CG,RATOM_NZ,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HE2,RATOM_HE3,RATOM_HG2,RATOM_HG3,RATOM_HZ1,RATOM_HZ2,RATOM_HZ3,RATOM_MAX},
 {-0.34810,-0.29030,0.84880,-0.82520,-0.05380,-0.03920,-0.01760,0.02270,-0.37410,-
0.82520,0.27640,0.14380,0.04820,0.04820,0.06110,0.06110,0.11210,0.11210,0.01340,0.01340,0.33740,0.33740,0.33740,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.8240,1.6612,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,1.1000,1.1000,1.4870,1.4870,0.6000,0.6000,0.6000,0},
```

```
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.1700,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,6,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,-1,-1,-1,-1,9,-1,-1,10,-1,-1,-1,11,-1,-1,-1,-1,12,13,-1,-1,-1,-1,14,-1,-1,-1,15,-1,-1,16,-1,-1,17,-1,-1,-1,-1,-1,18,-1,-1,-1,19,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,20,21,22,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Methionine coo- anion","MET",18, //21
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CE,RATOM_CG,RATOM_OXT,RATOM_SD,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HE1,RATOM_HE2,RATOM_HE3,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {-0.38210,-0.25970,0.80130,-0.81050,-0.02360,-0.03760,0.04920,-0.81050,-0.26920,0.26810,0.12770,0.04800,0.04800,0.06250,0.06250,0.06250,0.03170,0.03170,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,2.0000,0.6000,1.3870,1.4870,1.4870,1.3870,1.3870,1.3870,1.3870,1.3870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.2100,0.2500,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,5,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,8,-1,9,-1,-1,-1,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,13,14,-1,-1,15,-1,-1,-1,-1,-1,16,-1,-1,-1,17,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Phenylalanine coo- anion","PHE",21, //22
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CE1,RATOM_CE2,RATOM_CG,RATOM_CZ,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_HZ,RATOM_MAX},
 {-0.38210,-0.18250,0.76600,-0.80260,-0.09590,-0.13000,-0.13000,-0.18470,-0.18470,0.05520,-0.09440,-
0.80260,0.26810,0.10980,0.04430,0.04430,0.14080,0.14080,0.14610,0.14610,0.12800,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,1.4590,1.4590,1.4590,1.4590,1.4590,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0150,0.0150,0.0150,0},
 {0,1,2,3,4,-1,5,6,-1,7,8,-1,9,-1,-1,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,11,-1,-1,12,-1,-1,-1,13,-1,-1,-1,-1,14,15,16,-1,-1,-1,17,-1,-1,-1,-1,-1,18,19,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,20,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Proline coo- anion","PRO",15, //23
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_OXT,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {-0.28020,-0.13360,0.66310,-0.76970,-0.05430,0.04340,0.04660,-0.76970,0.07760,0.03810,0.03810,0.03310,0.03310,0.01720,0.01720,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,1.3870,1.4870,1.4870,1.3870,1.3870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,-1,9,10,-1,-1,-1,-1,11,-1,-1,-1,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,13,-1,-1,-1,14,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Serine coo- anion","SER",12, //24
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_OG,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HG,RATOM_MAX},
 {-0.38210,-0.27220,0.81130,-0.81320,0.11230,-0.65140,-0.81320,0.26810,0.13040,0.08130,0.08130,0.44740,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.7210,1.6612,0.6000,1.3870,1.3870,1.3870,0.0000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2104,0.2100,0.0157,0.0157,0.0157,0.0157,0.0000,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,6,-1,-1,7,-1,-1,-1,8,-1,-1,-1,-1,9,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Threonine coo- anion","THR",15, //25
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG2,RATOM_OG1,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB,RATOM_HG1,RATOM_HG21,RATOM_HG22,RATOM_HG23,RATOM_MAX},
 {-0.38210,-0.24200,0.78100,-0.80440,0.30250,-0.18530,-0.64960,-0.80440,0.26810,0.12070,0.00780,0.41190,0.05860,0.05860,0.05860,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.7210,1.6612,0.6000,1.3870,1.3870,0.0000,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.2104,0.2100,0.0157,0.0157,0.0157,0.0000,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,6,-1,7,-1,-1,8,-1,-1,-1,9,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,11,-1,-1,-1,-1,12,13,14,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Tryptophan coo- anion","TRP",25, //26
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CE2,RATOM_CE3,RATOM_CG,RATOM_CH2,RATOM_CZ2,RATOM_CZ3,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HE1,RATOM_HE3,RATOM_HH2,RATOM_HZ2,RATOM_HZ3,RATOM_NE1,RATOM_MAX},
 {-0.38210,-0.20840,0.76580,-0.80110,-0.07420,-0.18080,0.10780,0.12220,-0.18370,-0.07960,-0.10200,-0.25940,-0.22870,-
0.80110,0.26810,0.12720,0.04970,0.04970,0.20430,0.34130,0.14910,0.14010,0.15670,0.15070,-0.33160,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,1.4090,0.6000,1.4590,1.4590,1.4590,1.4590,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0150,0.0157,0.0150,0.0150,0.0150,0.0150,0.1700,0},
 {0,1,2,3,4,-1,5,6,-1,-1,7,8,9,-1,-1,10,-1,-1,11,12,-1,-1,-1,-1,-1,-1,-1,-1,13,-1,-1,14,-1,-1,-1,15,-1,-1,-1,-1,16,17,18,-1,-1,-1,-1,-1,-1,-1,-1,-1,19,-1,-1,-1,20,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,21,-1,-1,-1,-1,-1,-1,-1,-1,-1,22,23,-1,-1,-1,24,-1,-1,-1,-1}},
{"Tyrosine coo- anion","TYR",22, //27
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CE1,RATOM_CE2,RATOM_CG,RATOM_CZ,RATOM_OH,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_HH,RATOM_MAX},
 {-0.38210,-0.20150,0.78170,-0.80700,-0.07520,-0.19220,-0.19220,-0.24580,-0.24580,0.02430,0.33950,-0.56430,-
0.80700,0.26810,0.10920,0.04900,0.04900,0.17800,0.17800,0.16730,0.16730,0.40170,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.7210,1.6612,0.6000,1.3870,1.4870,1.4870,1.4590,1.4590,1.4590,1.4590,0.0000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.2104,0.2100,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0150,0.0150,0.0000,0},
 {0,1,2,3,4,-1,5,6,-1,7,8,-1,9,-1,-1,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,11,12,-1,-1,13,-1,-1,-1,14,-1,-1,-1,-1,15,16,17,-1,-1,-1,18,-1,-1,-1,-1,-1,19,20,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,21,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Valine coo- anion","VAL",17, //28
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG1,RATOM_CG2,RATOM_OXT,RATOM_H,RATOM_HA,RATOM_HB,RATOM_HG11,RATOM_HG12,RATOM_HG13,RATOM_HG21,RATOM_HG22,RATOM_HG23,RATOM_MAX},
 {-0.38210,-0.34380,0.83500,-0.81730,0.19400,-0.30640,-0.30640,-0.81730,0.26810,0.14380,0.03080,0.08360,0.08360,0.08360,0.08360,0.08360,0.08360,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,0.6000,1.3870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,5,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,8,-1,-1,-1,9,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,11,12,13,-1,14,15,16,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}}
};
```

residuesnt.cc

```
ResidueInfo ResidueInfoDatant[RESIDUEKEY_MAX] ={
{"Unavailable","ACE",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Alanine","ALA",12, //1
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB1,RATOM_HB2,RATOM_HB3,RATOM_MAX},
 {0.14140,0.09620,0.61630,-0.57220,-0.05970,0.19970,0.19970,0.19970,0.08890,0.03000,0.03000,0.03000,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,6,7,8,-1,-1,-1,9,10,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Arginine","ARG",26, //2
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_CZ,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HE,RATOM_HG2,RATOM_HG3,RATOM_HH11,RATOM_HH12,RATOM_HH21,RATOM_HH22,RATOM_NE,RATOM_NH1,RATOM_NH2,RATOM_MAX},
 {0.13050,-0.02230,0.72140,-
0.60130,0.01180,0.09350,0.02360,0.82810,0.20830,0.20830,0.20830,0.12420,0.02260,0.02260,0.05270,0.05270,0.35920,0.03090,0.03090,0.44940,0.44940,0.44940,0.44940,-0.56500,-
0.86930,-0.86930,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.3870,1.3870,0.6000,1.4870,1.4870,0.6000,0.6000,0.6000,0.6000,1.8240,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.1700,0.1700,0.1700,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,9,10,11,-1,-1,-1,-1,12,13,-1,-1,-1,-1,14,-1,-1,-1,15,16,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,17,-1,-1,-1,18,-
1,19,20,-1,21,22,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,23,-1,-1,24,25,-1}},
```

```
{"Unavailable","ASH",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Asparagine","ASN",16, //4
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG,RATOM_OD1,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD21,RATOM_HD22,RATOM_ND2,RATOM_MAX},
 {0.18010,0.03680,0.61630,-0.57220,-0.02830,0.58330,-0.57440,0.19210,0.19210,0.19210,0.12310,0.05150,0.05150,0.40970,0.40970,-0.86340,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.6612,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,0.6000,0.6000,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.1700,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,8,9,10,-1,-1,-1,-1,11,12,-1,-1,-1,-1,-1,13,14,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,15,-1,-1,-1,-1,-1,-1}},
{"Aspartic acid","ASP",14, //5
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG,RATOM_OD1,RATOM_OD2,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_MAX},
 {0.07820,0.02920,0.56210,-0.58890,-0.02350,0.81940,-0.80840,-0.80840,0.22000,0.22000,0.22000,0.11410,-0.01690,-0.01690,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.6612,1.6612,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.2100,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,6,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,9,10,11,-1,-1,-1,-1,12,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Unavailable","CYH",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Cysteine","CYS",13, //7
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_SG,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HSG,RATOM_MAX},
 {0.13250,0.09270,0.61230,-0.57130,-0.11950,-0.32980,0.20230,0.20230,0.20230,0.14110,0.11880,0.11880,0.19750,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,2.0000,0.6000,0.6000,0.6000,1.1000,1.3870,1.3870,0.6000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2500,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,6,7,8,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Cystine(s-s bridge)","CYX",12, //8
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_SG,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_MAX},
 {0.20690,0.10550,0.61230,-0.57130,-0.02770,-0.09840,0.18150,0.18150,0.18150,0.09220,0.06800,0.06800,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,2.0000,0.6000,0.6000,0.6000,1.1000,1.3870,1.3870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2500,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,6,7,8,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Unavailable","GLH",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Glutamine","GLN",19, //10
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_OE1,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HE21,RATOM_HE22,RATOM_HG2,RATOM_HG3,RATOM_NE2,RATOM_MAX},
 {0.14930,0.05360,0.61230,-0.57130,0.06510,0.73540,-0.09030,-0.61330,0.19960,0.19960,0.19960,0.10150,0.00500,0.00500,0.44290,0.44290,0.03310,0.03310,-1.00310,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,0.6000,0.6000,1.4870,1.4870,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.1094,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.1700,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,8,9,10,11,-1,-1,-1,-1,12,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,14,15,-1,-1,-1,-1,-1,-1,16,-1,-1,-1,17,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,18,-1,-1,-1}},
{"Glutamic acid","GLU",17, //11
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_OE1,RATOM_OE2,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {0.00170,0.05880,0.56210,-0.58890,0.09090,0.80870,-0.02360,-0.81890,-0.81890,0.23910,0.23910,0.23910,0.12020,-0.02320,-0.02320,-0.03150,-0.03150,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.6612,1.6612,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.1094,0.2100,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,8,-1,-1,-1,-1,-1,-1,-1,9,10,11,12,-1,-1,-1,-1,13,14,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,15,-1,-1,-1,16,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Glycine","GLY",9, //12
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA2,RATOM_HA3,RATOM_MAX},
 {0.29430,-0.01000,0.61630,-0.57220,0.16420,0.16420,0.16420,0.08950,0.08950,0},
 {1.8240,1.9080,1.9080,1.6612,0.6000,0.6000,0.6000,1.1000,1.1000,0},
 {0.1700,0.1094,0.0860,0.2100,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,4,5,6,-1,7,8,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Histidine deltah","HID",19, //13
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {0.15420,0.09640,0.61230,-0.57130,0.02590,0.10460,0.21270,-0.03990,0.19630,0.19630,0.19630,0.09580,0.02090,0.02090,0.36320,0.12990,0.13850,-0.38190,-0.57110,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,0.6000,1.4090,1.3590,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,9,10,11,-1,-1,-1,-1,12,13,14,-1,-1,-1,15,-1,-1,-1,-1,-1,16,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,17,-1,-1,-1,18,-1,-1,-1}},
{"Histidine epsilonh","HIE",19, //14
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {0.14720,0.02360,0.61230,-0.57130,0.04890,-0.23490,0.18040,0.17400,0.20160,0.20160,0.20160,0.13800,0.02230,0.02230,0.19630,0.13970,0.33240,-0.55790,-0.27810,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4090,1.3590,0.6000,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0157,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,9,10,11,-1,-1,-1,-1,12,13,-1,-1,-1,-1,14,-1,-1,-1,-1,-1,15,16,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,17,-1,-1,-1,18,-1,-1,-1}},
{"Histidine plus","HIP",20, //15
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_ND1,RATOM_NE2,RATOM_MAX},
 {0.25600,0.05810,0.72140,-0.60130,0.04840,-0.14330,-0.00110,-0.02360,0.17040,0.17040,0.17040,0.10470,0.05310,0.05310,0.38210,0.24950,0.26450,0.39210,-0.15100,-0.17390,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,0.6000,1.4090,1.3590,0.6000,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0157,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,9,10,11,-1,-1,-1,-1,12,13,14,-1,-1,-1,15,-1,-1,-1,-1,-1,16,17,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,18,-1,-1,-1,19,-1,-1,-1}},
{"Histidine","HIS",19, //16
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD2,RATOM_CE1,RATOM_CG,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_ND1,RATOM_NE2,RATOM_MAX},
```

```
 {0.15420,0.09640,0.61230,-0.57130,0.02590,0.10460,0.21270,-0.03990,0.19630,0.19630,0.19630,0.09580,0.02090,0.02090,0.36320,0.12990,0.13850,-0.38190,-0.57110,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,0.6000,1.4090,1.3590,1.8240,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.1700,0.1700,0},
 {0,1,2,3,4,-1,-1,5,-1,6,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,9,10,11,-1,-1,-1,-1,12,13,14,-1,-1,-1,15,-1,-1,-1,-1,-1,16,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,17,-1,-1,-1,18,-1,-1,-1}},
{"Isoleucine","ILE",21, //17
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CG1,RATOM_CG2,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB,RATOM_HD11,RATOM_HD12,RATOM_HD13,RATOM_HG12,RATOM_HG13,RATOM_HG21,RATOM_HG22,RATOM_HG23,RATOM_MAX},
 {0.03110,0.02570,0.61230,-0.57130,0.18850,-0.09080,-0.03870,-
0.37200,0.23290,0.23290,0.23290,0.10310,0.02130,0.02260,0.02260,0.02260,0.02010,0.02010,0.09470,0.09470,0.09470,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,5,-1,-1,-1,-1,-1,-1,6,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,9,10,11,-1,-1,12,-1,-1,-1,-1,13,14,15,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,16,17,-1,18,19,20,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Leucine","LEU",21, //18
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CG,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD11,RATOM_HD12,RATOM_HD13,RATOM_HD21,RATOM_HD22,RATOM_HD23,RATOM_HG,RATOM_MAX},
 {0.10100,0.01040,0.61230,-0.57130,-0.02440,-0.41060,-0.41040,0.34210,0.21480,0.21480,0.21480,0.10530,0.02560,0.02560,0.09800,0.09800,0.09800,0.09800,0.09800,0.09800,-
0.03800,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,5,6,-1,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,9,10,11,-1,-1,-1,-1,12,13,-1,14,15,16,-1,17,18,19,-1,-1,-1,-1,-1,-1,-1,20,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Unavailable","LYN",0,
 {RATOM_MAX},
 {0},
 {0},
 {0},
 {-1}},
{"Lysine","LYS",24, //20
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CE,RATOM_CG,RATOM_NZ,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HE2,RATOM_HE3,RATOM_HG2,RATOM_HG3,RATOM_HZ1,RATOM_HZ2,RATOM_HZ3,RATOM_MAX},
 {0.09660,-0.00150,0.72140,-0.60130,0.02120,-0.06080,-0.01810,-0.00480,-
0.37640,0.21650,0.21650,0.21650,0.11800,0.02830,0.02830,0.06330,0.06330,0.11710,0.11710,0.01210,0.01210,0.33820,0.33820,0.33820,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.8240,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4870,1.4870,1.1000,1.1000,1.4870,1.4870,0.6000,0.6000,0.6000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.1094,0.1700,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,6,-1,-1,-1,7,-1,-1,-1,-1,-1,-1,-1,8,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,9,10,11,12,-1,-1,-1,-1,13,14,-1,-1,-1,-1,15,-1,-1,-1,16,-1,-1,17,-1,-1,18,-1,-1,-1,-1,-1,19,-1,-1,-1,20,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,21,22,23,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Methionine","MET",19, //21
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CE,RATOM_CG,RATOM_SD,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HE1,RATOM_HE2,RATOM_HE3,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {0.15920,0.02210,0.61230,-0.57130,0.08650,-0.03410,0.03340,-0.27740,0.19840,0.19840,0.19840,0.11160,0.01250,0.01250,0.05970,0.05970,0.05970,0.02920,0.02920,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,2.0000,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.3870,1.3870,1.3870,1.3870,1.3870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.2500,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,5,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,-1,-1,8,9,10,11,-1,-1,-1,-1,12,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,14,15,-1,-1,16,-1,-1,-1,-1,-1,17,-1,-1,-1,18,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Phenylalanine","PHE",22, //22
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CE1,RATOM_CE2,RATOM_CG,RATOM_CZ,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_HZ,RATOM_MAX},
 {0.17370,0.07330,0.61230,-0.57130,0.03300,-0.13920,-0.13910,-0.16020,-0.16030,0.00310,-
0.12080,0.19210,0.19210,0.19210,0.10410,0.01040,0.01040,0.13740,0.13740,0.14330,0.14330,0.13290,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4590,1.4590,1.4590,1.4590,1.4590,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0150,0.0150,0.0150,0},
 {0,1,2,3,4,-1,5,6,-1,7,8,-1,9,-1,-1,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,11,12,13,14,-1,-1,-1,-1,15,16,17,-1,-1,-1,18,-1,-1,-1,-1,-1,19,20,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,21,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Proline","PRO",16, //23
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD,RATOM_CG,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD2,RATOM_HD3,RATOM_HG2,RATOM_HG3,RATOM_MAX},
 {-0.20200,0.10000,0.52600,-0.50000,-0.11500,-0.01200,-0.12100,0.31200,0.31200,0.10000,0.10000,0.10000,0.10000,0.10000,0.10000,0.10000,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,0.6000,0.6000,1.1000,1.4870,1.4870,1.1000,1.1000,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,5,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,8,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,12,-1,-1,-1,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,14,-1,-1,-1,15,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Serine","SER",13, //24
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_OG,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HG,RATOM_MAX},
 {0.18490,0.05670,0.61630,-0.57220,0.25960,-0.67140,0.18980,0.18980,0.18980,0.07820,0.02730,0.02730,0.42390,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.7210,0.6000,0.6000,0.6000,1.1000,1.3870,1.3870,0.0000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.2104,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0000,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,6,7,8,9,-1,-1,-1,-1,10,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Threonine","THR",16, //25
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG2,RATOM_OG1,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB,RATOM_HG1,RATOM_HG21,RATOM_HG22,RATOM_HG23,RATOM_MAX},
 {0.18120,0.00340,0.61630,-0.57220,0.45140,-0.25540,-0.67640,0.19340,0.19340,0.19340,0.10870,-0.03230,0.40700,0.06270,0.06270,0.06270,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.7210,0.6000,0.6000,0.6000,1.1000,1.3870,0.0000,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.2104,0.0157,0.0157,0.0157,0.0157,0.0157,0.0000,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,6,-1,-1,-1,-1,-1,-1,7,8,9,10,-1,-1,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,12,-1,-1,-1,-1,13,14,15,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Tryptophan","TRP",26, //26
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CE2,RATOM_CE3,RATOM_CG,RATOM_CH2,RATOM_CZ2,RATOM_CZ3,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HE1,RATOM_HE3,RATOM_HH2,RATOM_HZ2,RATOM_HZ3,RATOM_NE1,RATOM_MAX},
 {0.19130,0.04210,0.61230,-0.57130,0.05430,-0.17880,0.11320,0.15750,-0.22650,-0.16540,-0.10800,-0.27100,-
0.20340,0.18880,0.18880,0.18880,0.11620,0.02220,0.02220,0.21950,0.34120,0.16460,0.14110,0.15890,0.14580,-0.34440,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4090,0.6000,1.4590,1.4590,1.4590,1.4590,1.8240,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0157,0.0150,0.0150,0.0150,0.0150,0.1700,0},
 {0,1,2,3,4,-1,5,6,-1,-1,7,8,9,-1,-1,10,-1,-1,11,12,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,13,14,15,16,-1,-1,-1,-1,17,18,19,-1,-1,-1,-1,-1,-1,-1,-1,-1,20,-1,-1,-1,21,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
1,-1,22,-1,-1,-1,-1,-1,-1,-1,-1,-1,23,24,-1,-1,-1,25,-1,-1,-1,-1}},
{"Tyrosine","TYR",23, //27
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CD1,RATOM_CD2,RATOM_CE1,RATOM_CE2,RATOM_CG,RATOM_CZ,RATOM_OH,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB2,RATOM_HB3,RATOM_HD1,RATOM_HD2,RATOM_HE1,RATOM_HE2,RATOM_HH,RATOM_MAX},
 {0.19400,0.05700,0.61230,-0.57130,0.06590,-0.20020,-0.20020,-0.22390,-0.22390,-0.02050,0.31390,-
0.55780,0.18730,0.18730,0.18730,0.09830,0.01020,0.01020,0.17200,0.17200,0.16500,0.16500,0.40010,0},
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.9080,1.7210,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4590,1.4590,1.4590,1.4590,0.0000,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.0860,0.0860,0.0860,0.0860,0.0860,0.0860,0.2104,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0150,0.0150,0.0150,0.0150,0.0000,0},
 {0,1,2,3,4,-1,5,6,-1,7,8,-1,9,-1,-1,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,11,-1,-1,-1,-1,-1,12,13,14,15,-1,-1,-1,-1,16,17,18,-1,-1,-1,19,-1,-1,-1,-1,-1,20,21,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,22,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}},
{"Valine","VAL",18, //28
 {RATOM_N,RATOM_CA,RATOM_C,RATOM_O,RATOM_CB,RATOM_CG1,RATOM_CG2,RATOM_H1,RATOM_H2,RATOM_H3,RATOM_HA,RATOM_HB,RATOM_HG11,RATOM_HG12,RATOM_HG13,RATOM_HG21,RATOM_HG22,RATOM_HG23,RATOM_MAX},
 {0.05770,-0.00540,0.61630,-0.57220,0.31960,-0.31290,-0.31290,0.22720,0.22720,0.22720,0.10930,-0.02210,0.07350,0.07350,0.07350,0.07350,0.07350,0.07350,0},
```

```
 {1.8240,1.9080,1.9080,1.6612,1.9080,1.9080,1.9080,0.6000,0.6000,0.6000,1.1000,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,1.4870,0},
 {0.1700,0.1094,0.0860,0.2100,0.1094,0.1094,0.1094,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0.0157,0},
 {0,1,2,3,4,-1,-1,-1,-1,-1,-1,-1,-1,5,6,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,7,8,9,10,-1,-1,11,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,12,13,14,-1,15,16,17,-1,-1,-1,-
1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}}
};
```

rotamerconvert.cc

```
// ***********************************************
// File: rotmerconvert
// purpose: program to convert textversion of rotamer library
// to faster binary version used in the programs
// author: Hans-Henrik Staerfeldt
// notes:

#include "rotamer.hh"
#include <stream.h>

int main( int argc, char *argv[]) {
   rotamerlibrary myrotlib;
   cout ≪ "Converting ascii rotamere library ";
   cout ≪ argv[1] ≪ " to binary file " ≪ argv[2] ≪ endl;
   myrotlib.convert(argv[1],argv[2]);
   myrotlib.report();
   return 0;
}
```

makerotamerfields.cc

```
// ***********************************************
// File: makerotamerfields
// purpose: program to generate rotamer fields for
// all residue types and allsidecain dependencies
// author: Hans-Henrik Staerfeldt
// notes:

#include "rotamer.hh"
#include "structure.hh"
#include "grid.hh"
#include <fstream.h>
#include <iostream.h>

#define RESOLUTION 0.4
#define SIZE 64

void usage( void ) {
   cerr ≪ "makerotamerfields <RES>" ≪ endl;
   exit(1);
}

int main( int argc, char *argv[]) {
   rotamerlibrary myrotlib;
   rotamerrecord rec;
   Protein prot;
   ifstream *pdbin;
   Matrix4D m;
   GridSet mygrid(RESOLUTION,SIZE);
   GridSet mygridc(RESOLUTION,SIZE);
   Vector3D gridcenter(RESOLUTION*SIZE/2.0,
                       RESOLUTION*SIZE/2.0,
                       RESOLUTION*SIZE/2.0);
   double r;
   int i;
   int phi=-999;
   int psi=-999;
   char fname[1024];
   char *filename=argv[1];

   if (argc≠2) usage();

   try{
      myrotlib.open("data/rotamer.lib");
      myrotlib.report();

      // load each type up and assign it a random rotamer
      pdbin = new ifstream(filename);

      *pdbin ≫ prot;                                   /* read residue file */
      // transform protein to have the
      // N atom at (0,0,0)
      // CA along the x-axis and
      // C defining the z-axis
      try {
         m.viewat(prot.residue[0].atom[ATOM_N].pos,
                  prot.residue[0].atom[ATOM_CA].pos,
                  prot.residue[0].atom[ATOM_C].pos);
      }catch( char *error) {
         cerr≪"Warning; "≪error≪endl;
      }

      prot.transform(m);
      prot.translate(gridcenter);

      r=prot.getdiameter(prot.residue[0].atom[ATOM_N].pos);

      cerr≪filename≪" diameter:"≪r≪endl;

      myrotlib.set_type(ResidueInfoData[prot.residue[0].type].key);

      cerr ≪ filename ≪ ":  ";

      for (i=0;i<myrotlib.get_count();i++) {

         cerr ≪ i ≪ ",";

         myrotlib.get_record(&rec);

         if (phi≠-999 && psi≠-999 && (rec.phi≠phi || rec.psi≠psi)) {
            // save collected data
            sprintf(fname,"%s.%d.%d",
                    ResidueInfoData[prot.residue[0].type].key,phi,psi);
            mygridc.save(fname);
            mygridc.clear();
            phi=rec.phi;
            psi=rec.psi;
         }
         // build residue according to chi values of record
         prot.residue[0].buildchi(rec.chi);

         // render the fields of this residue
         mygrid.clear(0.0,0.0);
         prot.render(mygrid);
         mygridc.merge(mygridc,mygrid);
      }

      cerr ≪ endl;

      // save grid
      cout ≪ mygridc.Shape;

      delete pdbin;

      return 0;
   }
   catch ( char *e ){
      cerr≪"Caught exception:  "≪e≪endl;
      return -1;
   }
}
```

mergepdb.cc

```
// ***********************************************
// File: mergepdb.cc
// purpose: program used to merge pdb files using specific
// 4d rotation/translation matrixes
// author: Hans-Henrik Staerfeldt
// notes:

#include "structure.hh"
#include "vector3d.hh"
#include <pdb++.h>

void usage(void) {
   fprintf(stderr,"Usage:  mergepdb pdbA.ent chainA 4drotA "
           "pdbB.ent chainB 4drotB> pdb.out.ent \n");
   exit(1);
}

void matrixload( char *filename, Matrix4D &rot) {
   double m11,m12,m13,m14;
   double m21,m22,m23,m24;
   double m31,m32,m33,m34;
   double m41,m42,m43,m44;
   ifstream s;

   s.open(filename);

   if (s.bad()) throw "unable to read matrix file";

   s≫m11;
   s≫m12;
   s≫m13;
   s≫m14;
   s≫m21;
   s≫m22;
   s≫m23;
   s≫m24;
   s≫m31;
   s≫m32;
   s≫m33;
   s≫m34;
```

```
    s>>m41;
    s>>m42;
    s>>m43;
    s>>m44;

    rot.set(m11,m12,m13,m14,
            m21,m22,m23,m24,
            m31,m32,m33,m34,
            m41,m42,m43,m44 );
    s.close();
    rot.flip();
}

int main( int argc, char *argv[]) {
    Protein protA;
    Protein protB;
    Protein prot;
    Protein tmp;
    char chainsA[256];
    char chainsB[256];
    char nchainsA[]="RSTUVWXYZ";
    char nchainsB[]="LMNOPQ";
    Matrix4D rotA;
    Matrix4D rotB;
    int i;

    if (argc!=7) usage();

    try {

        prot.clear();
        prot.load(argv[1]);
        strcpy(chainsA,argv[2]);
        for (i=0;chainsA[i];i++) {
            if (chainsA[i]=='_') chainsA[i]=' ';
            tmp.clear();
            tmp=prot.getchain(chainsA[i]); // get relevant chain
            tmp.rechain(chainsA[i],nchainsA[i]);
            protA.merge(tmp);
        }
        matrixload(argv[3],rotA);
        protA.transform(rotA);

        prot.clear();
        prot.load(argv[4]);
        strcpy(chainsB,argv[5]);
        for (i=0;chainsB[i];i++) {
            if (chainsB[i]=='_') chainsB[i]=' ';
            tmp.clear();
            tmp=prot.getchain(chainsB[i]); // get relevant chain
            tmp.rechain(chainsB[i],nchainsB[i]);
            protB.merge(tmp);
        }
        matrixload(argv[6],rotB);
        protB.transform(rotB);

        prot.clear();
        prot.merge(protA);
        prot.merge(protB);
        cout<<prot;
    }
    catch ( char *e ){
        cerr<<"Caught exception:  "<<e<<endl;
        return -1;
    }

}
```

optimizeparam.cc

```
// ************************************************
// File: optimizeparm.cc
// purpose: program to optimize caling parameters
// for the energy correlation functions
// author: Hans-Henrik Staerfeldt
// notes:

#include "grid.hh"
#include "gridset.hh"
#include "score.hh"
#include "field.hh"
#include "structure.hh"
#include "vector3d.hh"
#include "filter.hh"
#include <pdb++.h>
#include <sys/types.h>
#include <time.h>

#define MAXRESULT 10

void usage(void) {
    fprintf(stderr,"Usage:  optimizeparam SIZE "
            "complex.ent Rchain Lchain\n");
    exit(1);
}

double randrange(double from, double to){
    return from+drand48()*(to-from);
}

void makemeasure(LigandGridSet &C,
                 Vector3D correctvector,
                 double newLJfield_scaling,
                 double newDesol_scaling,
                 double newElec_scaling,
                 double newShape_scaling ) {
    static vector <resultpoint> resultv; // topscoring results
    static CGrid resultg(C.scale,C.size); // result grid
    Vector3D gridcenter((C.size*C.scale)/2.0,
                        (C.size*C.scale)/2.0,
                        (C.size*C.scale)/2.0);

    LJfield_scaling=newLJfield_scaling;
    Desol_scaling=newDesol_scaling;
    Elec_scaling=newElec_scaling;
    Shape_scaling=newShape_scaling;

    // now do the folding
    cerr<<"sum results.."<<endl;

    // scale the precalculated scores
    scoresum(C,resultg);

    // Result.save(fname);
    resultv.clear();

    filterresults(resultg,0,32,resultv);

    cout<<"Correct is at "<<correctvector;
    cout<<" dist:  "<<measureresults(resultg,correctvector,resultv);
    cout<<" LJ= "<<LJfield_scaling;
    cout<<" ,D= "<<Desol_scaling;
    cout<<" ,E= "<<Elec_scaling;
    cout<<" ,S= "<<Shape_scaling<<endl;

}


int main( int argc, char *argv[]) {
    int size;
    double scale;
    Protein receptorp; // receptor atoms
    Protein ligandp; // ligand atoms
    Protein complexp; // complex atoms
    Protein rcomplexp; // reconstructed complex
    Protein rligandp; // reconstructed ligand

    LigandGridSet ligandgs; // ligand gridset
    ReceptorGridSet receptorgs; // receptor gridset

    Vector3D maxvector; // result vector
    Vector3D correctvector;

    Vector3D ligand_center;
    double ligand_diam;
    Vector3D receptor_center;
    double receptor_diam;

    char *rchain;
    char *lchain;
    int cx,cy,cz;

    Vector3D gridcenter;
    ifstream sin;

    try {
        if (argc!=5)
          usage();

        size=atoi(argv[1]);

        // read complex
        cerr<<"reading complex"<<endl;
        sin.open(argv[2]);
        sin >> complexp;
        sin.close();
```

```
        rchain=argv[3];
        if (!*rchain) usage();

        lchain=argv[4];
        if (!*lchain) usage();

        complexp.buildhydrogen();

        receptorp=complexp.getchains(rchain);
        cerr<<"finding receptor surface"<<endl;
        receptorp.findsurface(1.7);

        ligandp=complexp.getchains(lchain);
        cerr<<"finding ligand surface"<<endl;
        ligandp.findsurface(1.7);

        receptor_center=receptorp.getcog();
        cerr<<"receptor cog:  "<<receptor_center<<endl;

        receptor_diam=receptorp.getdiameter(receptor_center);
        cerr<<"receptor diameter:  "<<receptor_diam<<endl;

        ligand_center=ligandp.getcog();
        cerr<<"ligand cog:  "<<ligand_center<<endl;

        ligand_diam=ligandp.getdiameter(ligand_center);
        cerr<<"ligand diameter:  "<<ligand_diam<<endl;

        cerr<<"correct offset:  "<<ligand_center-receptor_center<<endl;

        // center entry to fit grid
        scale=(ligand_diam+receptor_diam+12.0)/size;

        cerr<<"grid scale:  "<<scale<<endl;

        ligandgs.init(scale,size);

        receptorgs.init(scale,size);

        gridcenter.set((size*scale)/2.0,
                       (size*scale)/2.0,
                       (size*scale)/2.0);

        correctvector=ligand_center-receptor_center+gridcenter;

        cerr << "gridcenter:  "<<gridcenter<<endl;

        // center to rendering space
        receptor_center= gridcenter-receptor_center;
        cerr << "receptor translation:  "<<receptor_center<<endl;
        receptorp.translate(receptor_center);

        ligand_center= gridcenter-ligand_center;
        cerr << "ligand translation:  "<<ligand_center<<endl;
        ligandp.translate(ligand_center);

        // render field
        cerr<<"render receptor.."<<endl;
        receptorp.render(receptorgs);

        cerr<<"render ligand.."<<endl;
        ligandp.render(ligandgs);

        cerr<<"folding and scoring"<<endl;

        // fourier transform receptor
        receptorgs.ifft();

        // heres all the magic!
        scorecorr(receptorgs,ligandgs);
        cerr<<"Done"<<endl;

        double parDesol_scaling=Desol_scaling;
// double parLJfield_scaling=LJfield_scaling;
        double parElec_scaling=Elec_scaling;
        double parShape_scaling=Shape_scaling;

        cx=correctvector.x()/scale;
        cy=correctvector.y()/scale;
        cz=correctvector.z()/scale;

        cout<<"Correct scores; ";
#ifdef DO_LJfield
        cout<<" LJ= "<<ligandgs.Density(cx,cy,cz);
#endif
#ifdef DO_Desol
        cout<<" ,D= "<<ligandgs.DesolL(cx,cy,cz);
#endif
#ifdef DO_Elec
        cout<<" ,E= "<<ligandgs.Elecd(cx,cy,cz);
#endif
#ifdef DO_Shape
        cout<<" ,S= "<<ligandgs.ShapeL(cx,cy,cz)<<endl;
#endif
        double e,s;
        int es,ss;

        for (es=-1;es<2;es+=2) {
            for (e=-6;e<10;e+=1) {
                for (ss=-1;ss<2;ss+=2) {
                    for (s=-6;s<10;s+=1) {

                        parShape_scaling=pow(2.0,s)*ss;
                        parElec_scaling=pow(2.0,e)*es;

                        makemeasure(ligandgs,
                                    correctvector,
                                    LJfield_scaling,
                                    parDesol_scaling,
                                    parElec_scaling,
                                    parShape_scaling );
                    }
                }
            }
        }
    }
    catch ( char *e ){
        cerr<<"Caught exception:  "<<e<<endl;
        return -1;
    }
}
```

recursedocking.cc

```
// **************************************************
// File: recursedocking
// purpose: implementation of recursive branch-and-bound
// docking algorithm.
// author: Hans-Henrik Staerfeldt
// notes: still lacks the bound implementations.

#include "grid.hh"
#include "field.hh"
#include "score.hh"
#include "structure.hh"
#include "vector3d.hh"
#include "rotationspace.hh"
#include "filter.hh"
#include <pdb++.h>
#include <sys/types.h>
#include <time.h>

#define SIZE 128
#define SCALE 0.8
#define ROTSUB 3
#define MAXROT 7682

#define MAXRESULTS 128

//#define SIZE 64
//#define SCALE 1.6
//#define ROTSUB 2
//#define MAXROT 1000

void usage(void) {
    fprintf(stderr,"Usage:  docking receptor.ent ligand.ent\n");
    exit(1);
}

void makeleafgridset(const LigandGridSet &LigandGS,
                     Rotationsegment <LigandGridSet> &rots) {

    static Matrix4D m;
    static Vector3D vx,vy,vz;
    const static Vector3D vxo(1.0,0.0,0.0);
    const static Vector3D vyo(0.0,1.0,0.0);
    const static Vector3D vzo(0.0,0.0,1.0);
    double phi=0;
    double theta=0;
    double psi=0;

    if (rots.data==NULL) {

        // add rotated LigandGS to rotLigandGS
        rots.getvector(phi,theta,psi);

        cerr<<"rotation is:  "<<phi<<","<<theta<<","<<psi<<endl;
```

```
        // now do the rotation
        cerr<<"rotating ligand gridset.."<<endl;

        // Make rotation matrix from phi,theta & psi
        Rotationspace<LigandGridSet>::getmatrix(phi,theta,psi,m);

        // Establish rotated coordinate system
        vx=m*vxo;
        vy=m*vyo;
        vz=m*vzo;

        rots.data = new LigandGridSet(LigandGS.scale,LigandGS.size);
        if (!rots.data)
          throw "unable to allocate rotation space gridset";
        rots.data->rotate(LigandGS,vz,vy);
    }else{
        cerr<<"reusing previous gridset"<<endl;
    }

}
//
// This function recursively collects all leafs of the
// rotationspace in rotLigandGS, that is a sampling
// of maximal energies for all leaf rotations of LigandGS
void recursivecollect(const LigandGridSet &LigandGS,
                      LigandGridSet &rotLigandGS,
                      Rotationsegment <LigandGridSet> &rots) {
    int i;

    // warning; static variable in recursive function,
    // should be ok though, as theyre all temporary
    if (rots.isleaf()) {

        makeleafgridset(LigandGS,rots);
        rotLigandGS.merge(*rots.data);

    }else{
        if (rots.data==NULL) {
            cerr<<"merging gridset children"<<endl;
            rots.data = new LigandGridSet(LigandGS.scale,LigandGS.size);
            if (!rots.data)
              throw "unable to allocate rotation space gridset";
            for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++) {
                if (rots.getchild(i)!=NULL) {
                    recursivecollect(LigandGS,*rots.data,*rots.getchild(i));
                    rotLigandGS.merge(*rots.data);
                }
            }
        }else{
            cerr<<"reusing previous merged gridset"<<endl;
            rotLigandGS.merge(*rots.data);
        }
    }
}

void recursivedock(const ReceptorGridSet &ReceptorGSfft,
                   const LigandGridSet &LigandGS,
                   Rotationsegment <LigandGridSet> rots,
                   int &ridx,
                   vector <resultpoint> &bestresults) {

    static vector <resultpoint> boundresults;
    static CGrid ResultG(LigandGS.scale,LigandGS.size); // just one
    int i;

    if (rots.isleaf()) {

        // make this leaf if its not already made
        makeleafgridset(LigandGS,rots);

        cerr<<"scoring leaf.."<<endl;

        scorecorr(ReceptorGSfft,*rots.data);

        scoresum(*rots.data,ResultG);

        // after this, we never need *rots.data again
        delete rots.data;
        rots.data=NULL;

        // sort/filter results
        filterresults(ResultG, ridx, MAXRESULTS, bestresults);

        sortresults(bestresults);

        // only keep the best
        limitresults(MAXRESULTS,bestresults);

        ridx++;

    }else{
        for (i=0;i<ROTATIONSEGMENT_CHILDREN;i++) {

            if (rots.getchild(i)!=NULL) {

                // create merging of subleafs
                rots.data =
                  new LigandGridSet(LigandGS.scale,LigandGS.size);

                if (!rots.data)
                  throw "unable to allocate rotation space gridset";

                recursivecollect(LigandGS,
                                 *rots.data,
                                 *rots.getchild(i));

                cerr<<"scoring node.."<<endl;

                scorecorr(ReceptorGSfft,*rots.data);

                scoresum(*rots.data,ResultG);

                // after this, we never need *rots.data again
                delete rots.data;
                rots.data=NULL;

                boundresults.clear();

                filterresults(ResultG, ridx, 1, boundresults);

                if (bestresults.size()==0 ||
                     (ResultG(boundresults[i].idx).re >
                      bestresults[bestresults.size()].v)) {

                    // The bound is better than the worst in the best,
                    // we have to recurse down
                    recursivedock(ReceptorGSfft,LigandGS,
                                  *rots.getchild(i),
                                  ridx,bestresults);

                }else{
                    cerr<<"Bound:   "<<ResultG(boundresults[i].idx).re<<endl;
                    cerr<<"Lastbest:   "<<bestresults[bestresults.size()-1].v<<endl;
                    // The bound is worse than the last best, so we dont have to
                    // bother recursing down

                    ridx+=rots.getchild(i)->size();
                }
            }
        }
    }
}

int main( int argc, char *argv[]) {
    double scale;
    Protein ReceptorProtein;
    Protein LigandProtein;

    LigandGridSet LigandGS(SCALE,SIZE);
    ReceptorGridSet ReceptorGS(SCALE,SIZE);
    LigandGridSet LigandcGS(SCALE,SIZE); // curent rotated ligand gridset
    CGrid ResultG(SCALE,SIZE);

    vector <resultpoint> ResultList;
    Rotationspace <LigandGridSet> rspace;
    vector <Vector3D> angles;

    int ridx;

    Vector3D bb1;
    Vector3D bb2;

    Matrix4D rot;
    Vector3D vx;
    Vector3D vy;
    Vector3D vz;
    Vector3D vxo(1.0,0.0,0.0);
    Vector3D vyo(0.0,1.0,0.0);
    Vector3D vzo(0.0,0.0,1.0);
    ofstream outfs;
    Vector3D Ligand_center;
    double Ligand_diam;
    Vector3D Receptor_center;
    double Receptor_diam;

    Vector3D gridcenter;
    ifstream sprot;

    try {
```

```
        if (argc≠3)
            usage();

        // read complex
        sprot.open(argv[1]);
        if (sprot.bad()) throw "error opening file";
        sprot ≫ ReceptorProtein;
        sprot.close();

        ReceptorProtein.buildhydrogen();

        sprot.open(argv[2]);
        if (sprot.bad()) throw "error opening file";
        sprot ≫ LigandProtein;
        sprot.close();

        LigandProtein.buildhydrogen();

        Receptor_center=ReceptorProtein.getcog();
        cerr≪"receptor cog:  "≪Receptor_center≪endl;

        Receptor_diam=ReceptorProtein.getdiameter(Receptor_center);
        cerr≪"receptor diameter:  "≪Receptor_diam≪endl;

        Ligand_center=LigandProtein.getcog();
        cerr≪"ligand cog:  "≪Ligand_center≪endl;

        Ligand_diam=LigandProtein.getdiameter(Ligand_center);
        cerr≪"ligand diameter:  "≪Ligand_diam≪endl;

        // center entry to fit grid
        scale=(Ligand_diam+Receptor_diam+12.0)/SIZE;

        cerr≪"minimal grid scale:  "≪scale≪endl;

        cerr≪"current grid scale:  "≪SCALE≪endl;

        assert(scale<SCALE);

        gridcenter.set((SIZE*SCALE)/2.0,
                       (SIZE*SCALE)/2.0,
                       (SIZE*SCALE)/2.0);
        cerr ≪ "gridcenter:  "≪gridcenter≪endl;

        // center to rendering space
        Receptor_center= gridcenter-Receptor_center;
        cerr ≪ "receptor translation:  "≪Receptor_center≪endl;
        ReceptorProtein.translate(Receptor_center);

        Ligand_center= gridcenter-Ligand_center;
        cerr ≪ "ligand translation:  "≪Ligand_center≪endl;
        LigandProtein.translate(Ligand_center);

        // render field
        cerr≪"render receptor.."≪endl;
        ReceptorProtein.render(ReceptorGS);
        cerr≪"render ligand.."≪endl;
        LigandProtein.render(LigandGS);
        cerr≪"done"≪endl;

        cerr≪"subdividing rotation space.."≪endl;
        rspace.subdivide(ROTSUB);
        cerr≪rspace.size()≪" angles found"≪endl;
        rspace.collectleafs(angles);

        ReceptorGS.ifft();

        for (ridx=0;ridx<ROTATIONSPACE_CHILDREN;ridx++) {
            int dummy=0;
            recursivedock(ReceptorGS,LigandGS,
                          *rspace.getchild(ridx),dummy,ResultList);
        }

        // now we tried them all out, now sort the results
        sortresults(ResultList);
        showresults(LigandGS.Density,angles,ResultList);
    }
    catch ( char *e ){
        cerr≪"Caught exception:  "≪e≪endl;
        return -1;
    }
}
```

testace.cc

```
// **************************************************
// File: testace
// purpose: perform the simulated anealing used to
// optimize the density field approximation to
// the ACE measure
// author: Hans-Henrik Staerfeldt
// notes:

#include <sys/types.h>
#include <time.h>
#include <sys/time.h>

#include "atomiccontactenergy.hh"
#include "complex.hh"

Complex ace1[18];
Complex ace2[18];
Complex ace1b[18];
Complex ace2b[18];

double initialcorr=0;
double initialmeasure=1000;
double lastmeasure=1000;
double thismeasure=1000;
double bestmeasure=1000;
const int iterations=50000000;
const double range=0.1;

double drandrange(double from, double to){
    return from+drand48()*(to-from);
}

double lrandrange(int from, int to){
    return from+lrand48()%(to-from+1);
}

double correlation(void) {
    int i;
    int j;
    int count;
    double sum;
    double da;
    double db;
    double avga;
    double avgb;
    double deva;
    double devb;

    deva=0; avga=0;
    devb=0; avgb=0;
    count=0;
    for (i=0;i<ACE_MAX;i++){
        for (j=0;j<ACE_MAX;j++){
            avga+=ACEinteraction[i][j];
            avgb+=(ace1[i]*ace2[j]).re;
            count++;
        }
    }
    avga/=(double(count));
    avgb/=(double(count));

    count=0;
    for (i=0;i<ACE_MAX;i++){
        for (j=0;j<ACE_MAX;j++){
            da=(ACEinteraction[i][j]-avga);
            deva+=da*da;
            db=((ace1[i]*ace2[j]).re-avgb);
            devb+=db*db;
            count++;
        }
    }
    deva=sqrt(deva/(double(count-1)));
    devb=sqrt(devb/(double(count-1)));

    count=0;
    for (i=0;i<ACE_MAX;i++){
        for (j=0;j<ACE_MAX;j++){
            da=(ACEinteraction[i][j] -avga)/deva;
            db=((ace1[i]*ace2[j]).re -avgb)/devb;
            sum+=da*db;
            count++;
        }
    }
    return sqrt(sum/double(count));
}

double measure(void) {
    int i;
    int j;
    int count;
    double sum;
    double da;

    count=0;
```

```
    sum=0;
    for (i=0;i<ACE_MAX;i++){
        for (j=0;j<ACE_MAX;j++){
            da=(ACEinteraction[i][j]-(ace1[i]*ace2[j]).re);
            sum+=da*da;
            count++;
        }
    }
    sum=sqrt(sum/double(count));
    return sum;
}

void reportresult() {
    int i;

    cout<<"measure;"<<measure()<<" vs "<<initialmeasure<<endl;
    cout<<"correlation;"<<correlation()<<" vs "<<initialcorr<<endl;

    cout << "Complex ACEfield[ACE_MAXTYPE] = {"<<endl;
    for (i=0;i<ACE_MAX-1;i++)
      cout<<ace1b[i]<<","<<endl;
    i=ACE_MAX-1;
    cout<<ace1b[i]<<endl;
    cout<<"};"<<endl;

    cout << "Complex ACEdensity[ACE_MAXTYPE] = {"<<endl;
    for (i=0;i<ACE_MAX-1;i++)
      cout<<ace2b[i].re<<","<<endl;
    i=ACE_MAX-1;
    cout<<ace2b[i]<<endl;
    cout<<"};"<<endl;
}

int eval(int iteration) {
    int i;
    double done=double(iteration)/double(iterations);

    thismeasure=measure();
    if (thismeasure<bestmeasure) {
        for (i=0;i<ACE_MAX;i++) {
            ace2b[i]=ace2[i];
            ace1b[i]=ace1[i];
        }
        cerr<<" measure now "<<thismeasure<<" vs ";
        cerr<<initialmeasure<<" "<<done*100<<"%"<<endl;
        bestmeasure=thismeasure;
        lastmeasure=thismeasure;
        reportresult();
        return 0;
    }
    if (drandrange(0,1)>done+0.25) {
        lastmeasure=thismeasure;
// cerr<<"x:"<<thismeasure<<endl;
        return 0; //accept regardless
    }
    if (thismeasure<lastmeasure ) {
        lastmeasure=thismeasure;
// cerr<<"a:"<<thismeasure<<endl;
        return 0;
    }else{
// cerr<<"r:"<<thismeasure<<endl;
        return 1;
    }
}

void anealingsearch() {
    int i;
    int j;
    double change;
    Complex oldval;

    for (i=0;i<ACE_MAX;i++){
        ace1[i].re=0.0;
        ace1[i].im=0.0;
        for (j=0;j<ACE_MAX;j++) {
            ace1[i].re+=ACEinteraction[i][j];
        }
        ace1[i].re/=18.0;
    }

    for (i=0;i<ACE_MAX;i++) {
        ace2[i].re=1;
        ace2[i].im=0;
        ace2b[i]=ace2[i];
        ace1b[i]=ace1[i];
    }

    bestmeasure=initialmeasure=lastmeasure=measure();
    initialcorr=correlation();
    reportresult();

    for (i=0;i<iterations;i++) {
        int which;

        j=lrandrange(0,17);

        which = lrandrange(0,3);
        switch (which) {
         case 0:
            // 0: change ace1 re
            oldval=ace1[j];
            change=ace1[j].re*drandrange(-range,range);
            ace1[j].re+=change;
            if (eval(i)) ace1[j]=oldval;
            break;
         case 1:
            // 1: change ace2 re
            oldval=ace2[j];
            change=ace2[j].re*drandrange(-range,range);
            ace2[j].re+=change;
            if (eval(i)) ace2[j]=oldval;
            break;
         case 2:
            // 2: add/sub ace2 re
            oldval=ace2[j];
            change=drandrange(-range,range);
            ace2[j].re+=change;
            if (eval(i)) ace2[j]=oldval;
            break;
         case 3:
            // 3: add/sub ace1 re
            oldval=ace1[j];
            change=drandrange(-range,range);
            ace1[j].re+=change;
            if (eval(i)) ace1[j]=oldval;
            break;

         case 4:
            // 4: change ace1 im
            oldval=ace1[j];
            change=ace1[j].im*drandrange(-range,range);
            ace1[j].im+=change;
            if (eval(i)) ace1[j]=oldval;
            break;
         case 5:
            // 5: change ace2 im
            oldval=ace2[j];
            change=ace2[j].im*drandrange(-range,range);
            ace2[j].im+=change;
            if (eval(i)) ace2[j]=oldval;
            break;
         case 6:
            // 6: add/sub ace1 im
            oldval=ace1[j];
            change=drandrange(-range,range);
            ace1[j].im+=change;
            if (eval(i)) ace1[j]=oldval;
            break;
         case 7:
            // 7: add/sub ace2 im
            oldval=ace2[j];
            change=drandrange(-range,range);
            ace2[j].im+=change;
            if (eval(i)) ace2[j]=oldval;
            break;

         case 8:
            // 8: swap sign im
            ace1[j].im* =-1;
            ace2[j].im* =-1;
            if (eval(i)) {
                ace1[j].im* =-1;
                ace2[j].im* =-1;
            }
            break;
        }
    }

    reportresult();
}

void testvalues(){
    int i;
    int j;

    for (i=0;i<ACE_MAX;i++){
        for (j=0;j<ACE_MAX;j++){
            cout<<"i:"<<i<<" j:"<<j;
            cout<<" ACE[i,j]:"<<ACEinteraction[i][j];
            cout<<" A[i]*B[j]:"<<(ACEfield[i]*ACEdensity[j]);
            cout<<" diff:\t"<<ACEinteraction[i][j]-
```

```
            (ACEfield[i]*ACEdensity[j]).re<<endl;
      }
    }
}

void main( int argc, char *argv[] ) {
    anealingsearch();
// testvalues();
}
```

bounddocking.cc

```
// ************************************************
// File: bounddocking.cc
// purpose: test program performing bound docking for
// a complex
// author: Hans-Henrik Staerfeldt
// notes:

#include "grid.hh"
#include "field.hh"
#include "structure.hh"
#include "vector3d.hh"
#include "filter.hh"
#include "rotationspace.hh"
#include <pdb++.h>
#include <sys/types.h>
#include <time.h>

#define SIZE 128
#define SCALE 0.8
#define ROTSUB 3
#define MAXROT 7682

//#define SIZE 64
//#define SCALE 1.6
//#define ROTSUB 2
//#define MAXROT 1000

//#define SIZE 32
//#define SCALE 3.2
//#define ROTSUB 2
//#define MAXROT 1000

void usage(void) {
    fprintf(stderr,"Usage:  bounddocking complex.ent Rchain Lchain\n");
    exit(1);
}

int main( int argc, char *argv[]) {
    double scale;
    Protein ReceptorProtein;
    Protein LigandProtein;
    Protein ComplexProtein;

    LigandGridSet LigandGS(SCALE,SIZE);
    ReceptorGridSet ReceptorGS(SCALE,SIZE);

    // curent rotated ligand gridset
    LigandGridSet LigandcGS(SCALE,SIZE);
    CGrid ResultG(SCALE,SIZE);
    vector <resultpoint> ResultList;

    Rotationspace <int> rspace;

    vector <Vector3D> angles;

    int ridx;

    Vector3D bb1;
    Vector3D bb2;

    Matrix4D rot;
    Vector3D vx;
    Vector3D vy;
    Vector3D vz;
    Vector3D vxo(1.0,0.0,0.0);
    Vector3D vyo(0.0,1.0,0.0);
    Vector3D vzo(0.0,0.0,1.0);
    ofstream outfs;
    Vector3D Ligand_center;
    double Ligand_diam;
    Vector3D Receptor_center;
    double Receptor_diam;
    char Rchain;
    char Lchain;

    Vector3D gridcenter;
    ifstream sprot;

    try {

        if (argc!=4)
            usage();

        // read complex
        sprot.open(argv[1]);
        if (sprot.bad()) throw "error opening file";
        sprot >> ComplexProtein;
        sprot.close();

        ComplexProtein.buildhydrogen();

        Rchain=argv[2][0];
        if (!Rchain) usage();
        if (Rchain=='_') Rchain=' ';

        Lchain=argv[3][0];
        if (!Lchain) usage();
        if (Lchain=='_') Lchain=' ';

        ReceptorProtein=ComplexProtein.getchain(Rchain);
        LigandProtein=ComplexProtein.getchain(Lchain);

        Receptor_center=ReceptorProtein.getcog();
        cerr<<"receptor cog:  "<<Receptor_center<<endl;

        Receptor_diam=ReceptorProtein.getdiameter(Receptor_center);
        cerr<<"receptor diameter:  "<<Receptor_diam<<endl;

        Ligand_center=LigandProtein.getcog();
        cerr<<"ligand cog:  "<<Ligand_center<<endl;

        Ligand_diam=LigandProtein.getdiameter(Ligand_center);
        cerr<<"ligand diameter:  "<<Ligand_diam<<endl;

        cerr<<"correct offset:  "<<Receptor_center-Ligand_center<<endl;

        // center entry to fit grid
        scale=(Ligand_diam+Receptor_diam+12.0)/SIZE;

        cerr<<"minimal grid scale:  "<<scale<<endl;

        cerr<<"current grid scale:  "<<SCALE<<endl;

        assert(scale<SCALE);

        gridcenter.set((SIZE*SCALE)/2.0,(SIZE*SCALE)/2.0,(SIZE*SCALE)/2.0);
        cerr << "gridcenter:  "<<gridcenter<<endl;

        // center to rendering space
        Receptor_center= gridcenter-Receptor_center;
        cerr << "receptor translation:  "<<Receptor_center<<endl;
        ReceptorProtein.translate(Receptor_center);

        Ligand_center= gridcenter-Ligand_center;
        cerr << "ligand translation:  "<<Ligand_center<<endl;
        LigandProtein.translate(Ligand_center);

        // render field
        cerr<<"render receptor.."<<endl;
        ReceptorProtein.render(ReceptorGS);
        cerr<<"render ligand.."<<endl;
        LigandProtein.render(LigandGS);
        cerr<<"done"<<endl;

        cerr<<"subdividing rotation space.."<<endl;
        rspace.subdivide(ROTSUB);
        cerr<<rspace.size()<<" angles found"<<endl;
        rspace.collectleafs(angles);

        for (ridx=0;ridx<angles.size();ridx++) {
            double phi,theta,psi;

            phi=angles[ridx].x();
            theta=angles[ridx].y();
            psi=angles[ridx].z();

            cerr<<"rotation "<<ridx<<"/"<<angles.size()<<" is:  ";
            cerr<<phi<<","<<theta<<","<<psi<<endl;

            // now do the rotation
            cerr<<"rotating ligand gridset.."<<endl;

            // Build coordinate system

            // Make rotation matrix from phi,theta & psi
```

```
            Rotationspace<int>::getmatrix(phi,theta,psi,rot);

            // Establish rotated coordinate system
            vx=rot*vxo;
            vy=rot*vyo;
            vz=rot*vzo;

            LigandcGS.rotate(LigandGS,vz,vy);

            // Now do the folding
            cerr<<"scoring.."<<endl;

            score(ReceptorGS,LigandcGS,ResultG);

            // sort/filter results
            filterresults(ResultG, ridx, SIZE, ResultList);

        }

        // now we tried them all out, now sort the results
        cerr<<"Sorting top scores;";
        sortresults(ResultList);

        showresults(ResultG,angles,ResultList);

    }
    catch ( char *e ){
        cerr<<"Caught exception:  "<<e<<endl;
        return -1;
    }
}
```

docking.cc

```
// ************************************************
// File: dokcing.cc
// purpose: simplified docking algorithm
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "grid.hh"
#include "field.hh"
#include "structure.hh"
#include "vector3d.hh"
#include "rotationspace.hh"
#include "filter.hh"
#include <pdb++.h>
#include <sys/types.h>
#include <time.h>

#define SIZE 128
#define SCALE 0.8
#define ROTSUB 3
#define MAXROT 7682

#define BESTLIST 4096

//#define SIZE 64
//#define SCALE 1.6
//#define ROTSUB 2
//#define MAXROT 1000

void usage(void) {
    fprintf(stderr,"Usage:  docking receptor.ent ligand.ent\n");
    exit(1);
}

int main( int argc, char *argv[]) {
    double scale;
    Protein ReceptorProtein;
    Protein LigandProtein;

    LigandGridSet LigandGS(SCALE,SIZE);
    ReceptorGridSet ReceptorGS(SCALE,SIZE);

    // curent rotated ligand gridset
    LigandGridSet LigandcGS(SCALE,SIZE);
    CGrid ResultG(SCALE,SIZE);

    vector <resultpoint> ResultList;
    Rotationspace <int> rspace;
    vector <Vector3D> angles;

    int ridx;

    Vector3D bb1;
    Vector3D bb2;
    Matrix4D rot;
    Vector3D vx;
    Vector3D vy;
    Vector3D vz;
    Vector3D vxo(1.0,0.0,0.0);
    Vector3D vyo(0.0,1.0,0.0);
    Vector3D vzo(0.0,0.0,1.0);
    ofstream outfs;
    Vector3D Ligand_center;
    double Ligand_diam;
    Vector3D Receptor_center;
    double Receptor_diam;

    Vector3D gridcenter;
    ifstream sprot;

    try {

        if (argc!=3)
          usage();

        // read complex
        sprot.open(argv[1]);
        if (sprot.bad()) throw "error opening file";
        sprot >> ReceptorProtein;
        sprot.close();

        ReceptorProtein.buildhydrogen();

        sprot.open(argv[2]);
        if (sprot.bad()) throw "error opening file";
        sprot >> LigandProtein;
        sprot.close();

        LigandProtein.buildhydrogen();

        Receptor_center=ReceptorProtein.getcog();
        cerr<<"receptor cog:  "<<Receptor_center<<endl;

        Receptor_diam=ReceptorProtein.getdiameter(Receptor_center);
        cerr<<"receptor diameter:  "<<Receptor_diam<<endl;

        Ligand_center=LigandProtein.getcog();
        cerr<<"ligand cog:  "<<Ligand_center<<endl;

        Ligand_diam=LigandProtein.getdiameter(Ligand_center);
        cerr<<"ligand diameter:  "<<Ligand_diam<<endl;

        // center entry to fit grid
        scale=(Ligand_diam+Receptor_diam+12.0)/SIZE;

        cerr<<"minimal grid scale:  "<<scale<<endl;

        cerr<<"current grid scale:  "<<SCALE<<endl;

        assert(scale<SCALE);

        gridcenter.set((SIZE*SCALE)/2.0,
                       (SIZE*SCALE)/2.0,
                       (SIZE*SCALE)/2.0);
        cerr << "gridcenter:  "<<gridcenter<<endl;

        // center to rendering space
        Receptor_center= gridcenter-Receptor_center;
        cerr << "receptor translation:  "<<Receptor_center<<endl;
        ReceptorProtein.translate(Receptor_center);

        Ligand_center= gridcenter-Ligand_center;
        cerr << "ligand translation:  "<<Ligand_center<<endl;
        LigandProtein.translate(Ligand_center);

        // render field
        cerr<<"render receptor.."<<endl;
        ReceptorProtein.render(ReceptorGS);
        cerr<<"render ligand.."<<endl;
        LigandProtein.render(LigandGS);
        cerr<<"done"<<endl;

        cerr<<"subdividing rotation space.."<<endl;
        rspace.subdivide(ROTSUB);
        cerr<<rspace.size()<<" angles found"<<endl;

        ReceptorGS.ifft();



        for (ridx=0;ridx<angles.size();ridx++) {
            double phi,theta,psi;

            phi=angles[ridx].x();
```

```
            theta=angles[ridx].y();
            psi=angles[ridx].z();

            cerr<<"rotation "<<ridx<<"/"<<angles.size()<<" is:  ";
            cerr<<phi<<","<<theta<<","<<psi<<endl;

            // now do the rotation
            cerr<<"rotating ligand gridset.."<<endl;

            // Make rotation matrix from phi,theta & psi
            Rotationspace<int>::getmatrix(phi,theta,psi,rot);

            // Establish rotated coordinate system
            vx=rot*vxo;
            vy=rot*vyo;
            vz=rot*vzo;

            LigandcGS.rotate(LigandGS,vz,vy);

            // Now do the folding
            cerr<<"scoring.."<<endl;

            scorecorr(ReceptorGS,LigandcGS);

            scoresum(LigandcGS,ResultG);

            // sort/filter results
            filterresults(ResultG, ridx, SIZE, ResultList);
        }

        // now we tried them all out, now sort the results

        sortresults(ResultList);
        showresults(ResultG,angles,ResultList);
    }
    catch ( char *e ){
        cerr<<"Caught exception:  "<<e<<endl;
        return -1;
    }
}
```

dumpgrid.cc

```
// ************************************************
// File: dumpgrid.cc
// purpose: dump contents of complex grid
//
// author: Hans-Henrik Staerfeldt
// notes:

#include <iostream.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <assert.h>
#include "grid.hh"
#include "complex.hh"

#define ABS(x) (((x)>=0)?(x):-(x))

void usage( void ){
    cerr<<"Usage:  dumpgrid gridfile "<<endl;
    exit(1);
}

int main(int argc, char *argv[]) {
    CGrid grid;
    int x,y,z,i;
    Complex max,min,avg,dev,dev0;
    double maxamp,minamp,avgamp,devamp,devamp0;
    ifstream *grids; // open grid stream

    // read grid
    cerr << "Loading :  " << argv[1] << endl;
    grids= new ifstream(argv[1]);
    (*grids) >> grid;

    grid.getstat(min,max,avg,dev,dev0,
                 minamp, maxamp, avgamp, devamp, devamp0);

    cerr<<"min:  "<<min<<endl;
    cerr<<"max:  "<<max<<endl;
    cerr<<"avg:  "<<avg<<endl;
    cerr<<"dev:  "<<dev<<endl;
    cerr<<"dev0:  "<<dev0<<endl;
    cerr<<"minamp:  "<<minamp<<endl;
    cerr<<"maxamp:  "<<maxamp<<endl;
    cerr<<"avgamp:  "<<avgamp<<endl;
    cerr<<"devamp:  "<<devamp<<endl;
    cerr<<"devamp0:  "<<devamp0<<endl;

    if (grid.getsizecubed()==0) {
        cerr << "Unable to find picture" << endl;
    }

    for (i=0,x=0;x<grid.getsize();x++) {

        // output header
        cout <<
"=====================================================";
        cout << "X slice "<<x<<endl;
        for (y=0;y<grid.getsize();y++) {
            cout << "Y row "<<y<<endl;
            for (z=0;z<grid.getsize();z++,i++) {
                cout<<grid(i);
            }
            cout<<endl;
        }
    }

    return 0;
}
```

filterpdb.cc

```
// ************************************************
// File: filterpdb.cc
// purpose: program implementing the filtering of the pdb
// datafiles
// author: Hans-Henrik Staerfeldt
// notes:

#include "structure.hh"
#include "vector3d.hh"
#include <pdb++.h>

void usage(void) {
    fprintf(stderr,"Usage:  filterpdb chains [rechain] < pdb.ent "
               "> pdb.out.ent \n");
    exit(1);
}

int main( int argc, char *argv[]) {
    Protein prot;
    Protein protf;
    char rechain='
0';
    char chains[256];
    if (argc!=2 && argc!=3) usage();
    if (argc==3) rechain=*argv[2];

    strcpy(chains,argv[1]);
    for (int i=0;chains[i];i++)
      if (chains[i]=='_') chains[i]=' ';

    try {
        cerr<<"reading"<<endl;
        cin >> prot; // read it.
// prot.translate(-prot.getcog()); // center it.
        protf=prot.getchains(chains); // get relevant chains
        if (rechain) protf.rechain(rechain); // rechain it for CE
        cerr<<"build hydrogen"<<endl;
        protf.buildhydrogen(); // build hydrogens
// cerr<<"find surface"<<endl;
// protf.findsurface(1.5);
        cerr<<"writing"<<endl;
        cout << protf; // hello! print it
    }
    catch ( char *e ){
        cerr<<"Caught exception:  "<<e<<endl;
        return -1;
    }

}
```

getslice.cc

```
// ************************************************
// File: getslice.cc
// purpose: program for making graph of data in a grid.
//
// author: Hans-Henrik Staerfeldt
// notes:
```

```
#include <iostream.h>
#include <stdlib.h>
#include "grid.hh"

#define ABS(x) (((x)≥0)?(x):-(x))

void usage( void ){
#ifdef RAWSLICE
    cerr≪"Usage:  getsliceraw [min max]< gridfile > datfile"≪endl;
#else
    cerr≪"Usage:  getslice [min max [argument]]< gridfile > datfile"≪endl;
#endif
    exit(1);
}

int main(int argc, char *argv[]) {
    CGrid a;
    int x,y,z,i;
    double sum;
    Complex v;
    double maxsum=0;
    double amin=-DBL_MAX;
    double amax=DBL_MAX;
    int maxslice=0;
#ifndef RAWSLICE
    char *arg=NULL;
    ofstream cpipe;
    FILE *pf;
#endif

    if (argc==3
#ifndef RAWSLICE
        || argc==4
#endif
         ) {
        amin=atof(argv[1]);
        amax=atof(argv[2]);

#ifndef RAWSLICE
        if (argc==4) arg=argv[3];
#endif

    }else if (argc≠1) usage();

    cin≫a;

    // find slice
    for (i=0,x=0;x<a.getsize();x++) {

        sum=0;

        for (y=0;y<a.getsize();y++)
          for (z=0;z<a.getsize();z++,i++) {
               v=a(i);
               // make limits
               if (v.re<amin) v.re=amin;
               if (v.re>amax) v.re=amax;
               if (v.im<amin) v.im=amin;
               if (v.im>amax) v.im=amax;
               a(i)=v;
               sum+=amplitude(v);
          }

        if (sum≥maxsum) {
            maxslice=x;
            maxsum=sum;
        }
    }

    if (maxsum==0) {
        cerr≪"Unable to find slice" ≪ endl;
        exit(1);
    }

#ifdef RAWSLICE
    a.writesliceregp(maxslice,cout);
#else
    pf=popen("/usr/local/bin/gnuplot-3.7","w");
    cpipe.attach(fileno(pf));
    cpipe≪"set term post eps enhanced color solid"≪endl;
    cpipe≪"set hidden3d"≪endl;
    cpipe≪"set contour"≪endl;
    cpipe≪"set view 40"≪endl;
    cpipe≪"set title 'maximal slice is at x=";
    cpipe≪maxslice*a.getscale()≪"'"≪endl;
    if (arg)
      cpipe≪arg≪endl;
    cpipe≪"splot '-' w steps"≪endl;
    a.writesliceregp(maxslice,cpipe);
    cpipe≪"e";
    cpipe.close();
    pclose(pf);
#endif

    return 0;
}
```

rendergrid.cc

```
// **************************************************
// File: rendergrid.cc
// purpose: function to render a grid into a series of
// graphical images for viewing.
// author: Hans-Henrik Staerfeldt
// notes: special color model is used to show
// both real and imaginary data in same image

#include <iostream.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "grid.hh"
#include "complex.hh"

#define ABS(x) (((x)≥0)?(x):-(x))

void usage( void ){
    cerr≪"Usage:  rendegrid gridfile "≪endl;
    exit(1);
}

void HSVtoRGB(double &r, double &g, double &b,
         double h, double s, double v ) {


    // H is given on [0, 6] or UNDEFINED. S and V are given on [0, 1].
    // RGB are each returned on [0, 1].
    double m, n, f;
    int i;

    if(h<0 ) {
        r=v;
        g=v;
        b=v;
    }

    h/=60; // Degrees → [0,6];

    i = floor(h);
    f = h - i;
    if(!(i & 1)) f = 1 - f; // if i is even
    m = v * (1 - s);
    n = v * (1 - s * f);
    switch (i) {
     case 6:
     case 0: r=v; g=n; b=m; break;
     case 1: r=n; g=v; b=m; break;
     case 2: r=m; g=v; b=n; break;
     case 3: r=m; g=n; b=v; break;
     case 4: r=n; g=m; b=v; break;
     case 5: r=v; g=m; b=n; break;
    }

}
// converting hue, saturation and brightness to RGB
void HSBtoRGB(double &r, double &g, double &b,
              double h, double s, double v )
{
    int i;
    double f, p, q, t;

    if( s == 0.0 ) {
        // achromatic (grey)
        r = g = b = v;
        return;
    }

    h /= 60.0; // sector 0 to 5
    i = floor( h );
    f = h - i; // factorial part of h
    p = v * ( 1 - s );
    q = v * ( 1 - s * f );
    t = v * ( 1 - s * ( 1 - f ) );

    switch( i ) {
     case 0:
```

```
        r = v;
        g = t;
        b = p;
        break;
      case 1:
        r = q;
        g = v;
        b = p;
        break;
      case 2:
        r = p;
        g = v;
        b = t;
        break;
      case 3:
        r = p;
        g = q;
        b = v;
        break;
      case 4:
        r = t;
        g = p;
        b = v;
        break;
      default: // case 5:
        r = v;
        g = p;
        b = q;
        break;
    }
}

// render amplitude as brightness/satuarion and phase as hue
void ComplextoRGB(double &r, double &g, double &b, const Complex &c) {
    double h,s,v;
    double p,rc,ac;
    static double sqrt2=sqrt(2.0);

#define SMALL (1.0e-10)

    ac=amplitude(c);
    v=1-(ac/sqrt2)*0.5;
    s=(ac/sqrt2);

    if (ac<SMALL)
      rc=1.0;
    else
      rc=c.re/ac;

    p=acos(rc);
    if (c.im<0) p=(2.0*PI)-p;

    h=(360.0*p)/(2.0*PI);

    assert(v>=0.0 && v<=1.0);
    assert(s>=0.0 && s<=1.0);
    assert(h>=0.0 && h<=360.0);

    HSVtoRGB(r,g,b,h,s,v);

    assert(r>=0.0 && r<=1.0);
    assert(g>=0.0 && g<=1.0);
    assert(b>=0.0 && b<=1.0);

// cerr<<"("<<c.re<<","<<c.im<<")->";
// cerr<<"("<<h<<","<<s<<","<<v<<")->";
// cerr<<"("<<r<<","<<g<<","<<b<<")"<<endl;
}

Complex normalize(const Complex &c, const double devamp){
    Complex r;
    double d;

    r.re=c.re/devamp;
    r.im=c.im/devamp;

    if (amplitude(r)>1.0) {
        // cut at amplitude 1.0
        d=amplitude(r);
        r.re/=d;
        r.im/=d;
    }

// cerr<<"("<<c.re<<","<<c.im<<")->";
// cerr<<"("<<r.re<<","<<r.im<<")";

    return r;
}

unsigned char cnv256(double v) {
    v* =256.0;
    return 255-floor(256.0-v);
}

int main(int argc, char *argv[]) {
    CGrid grid;
    int x,y,z,i;
    Complex max,min,avg,dev,dev0;
    double maxamp,minamp,avgamp,devamp,devamp0;

    double r,g,b;
    Complex c;
    char fname[1024];
    unsigned char *pixels;
    int pi;
    int argi;
    int gridsize=0;
    ifstream *grids; // open grid stream
    ofstream *ppms;

    argi=1;
    if (!strcmp(argv[argi],"-i")) {
        argc++;
    }

    sprintf(fname,"%s.ppm",argv[argi]);
    mkdir(fname,S_IRWXU|S_IRWXG);

    // read grid
    cerr << "Loading :  " << argv[argi] << endl;
    grids= new ifstream(argv[argi]);
    (*grids) >> grid;

    grid.getstat(min,max,avg,dev,dev0,
                    minamp, maxamp, avgamp, devamp, devamp0);

    cerr<<"min:   "<<min<<endl;
    cerr<<"max:   "<<max<<endl;
    cerr<<"avg:   "<<avg<<endl;
    cerr<<"dev:   "<<dev<<endl;
    cerr<<"dev0:   "<<dev0<<endl;
    cerr<<"minamp:   "<<minamp<<endl;
    cerr<<"maxamp:   "<<maxamp<<endl;
    cerr<<"avgamp:   "<<avgamp<<endl;
    cerr<<"devamp:   "<<devamp<<endl;
    cerr<<"devamp0:   "<<devamp0<<endl;

    // assume scaling symmetrical around 0
    if (grid.getsizecubed()==0 || devamp0==0) {
        cerr << "Unable to find picture" << endl;
        exit(0);
    }

    gridsize=grid.getsize();

    pixels=new unsigned char[gridsize*gridsize*3];

    for (i=0,x=0;x<grid.getsize();x++) {

        // collect pixels
        for (pi=0,y=0;y<grid.getsize();y++) {
            for (z=0;z<grid.getsize();z++,i++) {
                c=normalize(grid(i),devamp0);
                ComplextoRGB(r,g,b, c);
                pixels[pi++]=cnv256(r);
                pixels[pi++]=cnv256(g);
                pixels[pi++]=cnv256(b);
            }
        }

        // print slice
        sprintf(fname,"%s.ppm/%03d.ppm",argv[argi],x);
        cerr<<fname<<endl;
        ppms=new ofstream(fname);
        // output ppm header
        (*ppms)<<"P6"<<endl<<grid.getsize()<<" ";
        (*ppms)<<grid.getsize()<<endl<<"255"<<endl;
        (*ppms).write(pixels,gridsize*gridsize*3);
        delete ppms;
    }

    return 0;
}
```

testfft.cc

```
// ***************************************************
// File: testfft.cc
// purpose: test the folding ability of the discreet fourier transform
```

```
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "grid.hh"
#include "vector3d.hh"
#include <pdb++.h>

//#define ROTATEEULER

void usage(void) {
#ifdef ROTATEEULER
    fprintf(stderr,"Usage: rotatepdb phi theta psi "
            "< pdb.ent > pdb.rot.ent \n");
#else
    fprintf(stderr,"Usage: rotatepdb x y z theta "
            "< pdb.ent > pdb.rot.ent \n");
#endif
    exit(1);
}

int main( int argc, char *argv[]) {
    Protein prot;
    Matrix4D mat;

    try {
#ifdef ROTATEEULER
        // euler rotation
        double phi=0.0,theta=0.0,psi=0.0;

        if (argc≠4) usage();

        sscanf(argv[1],"%lf",&phi);
        sscanf(argv[2],"%lf",&theta);
        sscanf(argv[3],"%lf",&psi);

        mat.rotation(phi,theta,psi); // wow make a rotation matrix
#else

        // quaternion rotation
        Vector3D axis;
        double theta=0.0;
        Quaternion q;

        if (argc≠5) usage();

        sscanf(argv[3],"%lf",&axis[0]);
        sscanf(argv[2],"%lf",&axis[1]);
        sscanf(argv[1],"%lf",&axis[2]);
        sscanf(argv[1],"%lf",&theta);

        q.set(axis,theta);
        mat.rotation(q);

#endif

        cin ≫ prot; // oooh read it.

        prot.transform(mat); // coool, rotate it

        cout ≪ prot; // hello! print it
    }
    catch ( char *e ){
        cerr≪"Caught exception: "≪e≪endl;
        return -1;
    }

}
```

testfoldfilter.cc

```
// ************************************************
// File: testfoldfilter
// purpose: test filtering of results
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "grid.hh"
#include "field.hh"
#include "structure.hh"
#include "vector3d.hh"
#include "filter.hh"
#include <pdb++.h>
#include <sys/types.h>
#include <time.h>
```

```
#define MAXRESULT 10

void usage(void) {
    fprintf(stderr,"Usage: testfoldfilter complex.ent "
            "complex.ent.shape.fold.grid Rchain Lchain \n");
    exit(1);
}

int main( int argc, char *argv[]) {

    Protein receptorp;
    Protein ligandp;
    Protein complexp;
    CGrid Shape;
    ofstream outfs;
    vector <resultpoint> results;

    Vector3D ligand_center;
    double ligand_diam;
    Vector3D receptor_center;
    double receptor_diam;

    Vector3D maxvector;
    Vector3D gridcenter(1.0,1.0,1.0);
    char fname[1024];
    char *Rchain;
    char *Lchain;
    Protein rcomplexp;
    Protein rligandp;

    int i,idx;
    int x,y,z;

    ifstream ins;

    Matrix4D mat;

    try {

        if (argc≠5)
            usage();

        // read complex
        ins.open(argv[1]);
        if (ins.bad()) throw "could not open complex";
        ins ≫ complexp;
        ins.close();

        ins.open(argv[2]);
        if (ins.bad()) throw "could not open fold grid";
        ins ≫ Shape;
        ins.close();

        Rchain=argv[3];
        if (!*Rchain) usage();

        Lchain=argv[4];
        if (!*Lchain) usage();

        complexp.buildhydrogen();

        receptorp=complexp.getchains(Rchain);
        ligandp=complexp.getchains(Lchain);

        receptor_center=receptorp.getcog();
        cerr≪"receptor cog: "≪receptor_center≪endl;

        receptor_diam=receptorp.getdiameter(receptor_center);
        cerr≪"receptor diameter: "≪receptor_diam≪endl;

        ligand_center=ligandp.getcog();
        cerr≪"ligand cog: "≪ligand_center≪endl;

        ligand_diam=ligandp.getdiameter(ligand_center);
        cerr≪"ligand diameter: "≪ligand_diam≪endl;

        cerr≪"correct offset: "≪receptor_center-ligand_center≪endl;

        cerr≪"reading grid"≪endl;

        cerr≪"grid scale: "≪Shape.getscale()≪endl;
        gridcenter=gridcenter*((Shape.getsize()*Shape.getscale())/2.0);
        cerr≪"grid center: "≪gridcenter;

        // center to rendering space
        cerr ≪ "receptor translation: "≪gridcenter - receptor_center≪endl;
        receptorp.translate(gridcenter - receptor_center);
        cerr≪"grid center: "≪gridcenter;

        cerr ≪ "ligand translation: "≪gridcenter - ligand_center≪endl;
```

```
        ligandp.translate(gridcenter - ligand_center);

        // output field
        cerr<<"searching.."<<endl;
        for (i=0;i<Shape.getsizecubed();i++){
            Shape(i).im=0.0; // throw away, use again
            if (Shape(i).re<-Shape.getsize()*10)
              Shape(i).re=-Shape.getsize()*10;
        }

        filterresults(Shape,0,MAXRESULT,results);

        for (i=0;i<results.size();i++) {

            idx=results[i].idx;

            maxvector=Shape.getvector(idx);
            maxvector=maxvector-gridcenter;//compensate for centering

            Shape.getposition(idx,x,y,z);

            cerr << "Maximum at:"<<"(pos:"<<x<<","<<y<<","<<z<<")";
            cerr <<" (value:"<<results[i].v<<") "<<maxvector<<endl;

            // output merged pdb files.
            cerr<<"merging and writing result pdb file"<<endl;

            // relocate translation
            maxvector= Shape.getvector(x,y,z)-gridcenter;

            rcomplexp.clear();
            rligandp.clear();
            rligandp.merge(ligandp);
            rligandp.translate(maxvector);
            rcomplexp.merge(rligandp);
            rcomplexp.merge(receptorp);
            sprintf(fname,"%s.shape.dock.%d.ent",argv[2],i);
            rcomplexp.save(fname);
        }


    }
    catch ( char *e ){
        cerr<<"Caught exception:  "<<e<<endl;
        return -1;
    }
}
```

testfoldpdb.cc

```
// ************************************************
// File: testfoldpdb
// purpose: test the ability to sample individual
// receptor and linagd density and field functions and
// create correlation grids with ranked cells. Then
// write the merged protein structures for the found
// top scores.
// author: Hans-Henrik Staerfeldt
// notes:

#include "grid.hh"
#include "field.hh"
#include "structure.hh"
#include "vector3d.hh"
#include "filter.hh"
#include <pdb++.h>
#include <sys/types.h>
#include <time.h>

#define MAXRESULT 10

void usage(void) {
    fprintf(stderr,"Usage:  testfoldpdb SIZE complex.ent Rchain Lchain\n");
    exit(1);
}

double randrange(double from, double to){
    return from+drand48()*(to-from);
}

int main( int argc, char *argv[]) {
    int size;
    double scale;
    Protein receptorp; // receptor atoms
    Protein ligandp; // ligand atoms
    Protein complexp; // complex atoms
    Protein rcomplexp; // reconstructed complex
    Protein rligandp; // reconstructed ligand

    vector <resultpoint> resultv; // topscoring results

    LigandGridSet ligandgs; // ligand gridset
    ReceptorGridSet receptorgs; // receptor gridset
    CGrid resultg; // result grid

    Vector3D maxvector; // result vector
    int idx,x,y,z,i;

    Vector3D ligand_center;
    double ligand_diam;
    Vector3D receptor_center;
    double receptor_diam;
    int cx,cy,cz;

    char *rchain;
    char *lchain;
    char fname[1024];

    Vector3D gridcenter;
    ifstream sin;

    try {
        if (argc!=5)
          usage();

        size=atoi(argv[1]);

        // read complex
        cerr<<"reading complex"<<endl;
        sin.open(argv[2]);
        sin >> complexp;
        sin.close();

        rchain=argv[3];
        if (!*rchain) usage();

        lchain=argv[4];
        if (!*lchain) usage();

        complexp.buildhydrogen();

        receptorp=complexp.getchains(rchain);
        cerr<<"finding receptor surface"<<endl;
        receptorp.findsurface(1.7);
#ifdef DOSAVE
        sprintf(fname,"%s.1dock.R.ent",argv[2]);
        receptorp.save(fname);
#endif

        ligandp=complexp.getchains(lchain);
        cerr<<"finding ligand surface"<<endl;
        ligandp.findsurface(1.7);
#ifdef DOSAVE
        sprintf(fname,"%s.1dock.L.ent",argv[2]);
        ligandp.save(fname);
#endif

        receptor_center=receptorp.getcog();
        cerr<<"receptor cog:  "<<receptor_center<<endl;

        receptor_diam=receptorp.getdiameter(receptor_center);
        cerr<<"receptor diameter:  "<<receptor_diam<<endl;

        ligand_center=ligandp.getcog();
        cerr<<"ligand cog:  "<<ligand_center<<endl;

        ligand_diam=ligandp.getdiameter(ligand_center);
        cerr<<"ligand diameter:  "<<ligand_diam<<endl;

        cerr<<"correct offset:  "<<ligand_center-receptor_center<<endl;

        // center entry to fit grid
        scale=(ligand_diam+receptor_diam+12.0)/size;

        cx=((ligand_center-receptor_center).x()/scale)+(size/2);
        cy=((ligand_center-receptor_center).y()/scale)+(size/2);
        cz=((ligand_center-receptor_center).z()/scale)+(size/2);

        cerr<<"correct index:  "<<cx<<","<<cy<<","<<cz<<endl;

        cerr<<"grid scale:  "<<scale<<endl;

        ligandgs.init(scale,size);
        receptorgs.init(scale,size);

        resultg.init(scale,size);
        gridcenter.set((size*scale)/2.0,(size*scale)/2.0,(size*scale)/2.0);

        cerr << "gridcenter:  "<<gridcenter<<endl;
```

```
        // center to rendering space
        receptor_center= gridcenter-receptor_center;
        cerr ≪ "receptor translation:  "≪receptor_center≪endl;
        receptorp.translate(receptor_center);

        ligand_center= gridcenter-ligand_center;
        cerr ≪ "ligand translation:  "≪ligand_center≪endl;
        ligandp.translate(ligand_center);

        // render field
        cerr≪"render receptor.."≪endl;
        receptorp.render(receptorgs);

#ifdef DOSAVE
        cerr≪"writing result"≪endl;
        sprintf(fname,"%s.R.ldock",argv[2]);
        receptorgs.save(fname);
#endif

        cerr≪"render ligand.."≪endl;
        ligandp.render(ligandgs);

#ifdef DOSAVE
        cerr≪"writing result"≪endl;
        sprintf(fname,"%s.R.ldock",argv[2]);
        ligandgs.save(fname);
#endif
        cerr≪"Done"≪endl;

        // now do the folding
        cerr≪"folding.."≪endl;

#define TRY1
#ifdef TRY1
        // heres all the magic!
        score(receptorgs,ligandgs,resultg);
        // Result.save(fname);
#else
        receptorgs.ifft();
        scorecorr(receptorgs,ligandgs);
        scoresum(ligandgs,resultg);
#endif

        filterresults(resultg,0,MAXRESULT,resultv);

        for (i=0;i<resultv.size();i++) {

            idx=resultv[i].idx;

            resultg.getposition(idx,x,y,z);

            maxvector=resultg.getvector(x,y,z);

            maxvector=(maxvector-gridcenter);//compensate for centering
            cerr ≪ "Maximum at:"≪"(pos:"≪x≪","≪y≪","≪z≪")";
            cerr ≪" (value:"≪resultv[i].v≪") "≪maxvector≪endl;

            // output merged pdb files.
            cerr≪"merging and writing result pdb file"≪endl;

            // relocate translation

            rcomplexp.clear();
            rligandp.clear();
            rligandp.merge(ligandp);
            rligandp.translate(maxvector);
            rcomplexp.merge(rligandp);
            rcomplexp.merge(receptorp);
            sprintf(fname,"%s.dock.%d.ent",argv[2],i);
            rcomplexp.save(fname);
        }

    }
    catch ( char *e ){
        cerr≪"Caught exception:  "≪e≪endl;
        return -1;
    }
}
```

testgrid.cc

```
// ***************************************************
// File: testgrid.cc
// purpose: test the CGrid class
//
// author: Hans-Henrik Staerfeldt
// notes:

#include <stdlib.h>
#include <time.h>
#include <fstream.h>
#include "grid.hh"
#include "field.hh"
#include "vector3d.hh"

#define SIZE 64
#define RESO 0.1

void usage ( void ) {
    cerr≪"Usage:  testgrid testfile1 testfile2"≪endl;
    exit(1);
}
//
// test1: test that writing/reading works.
// test2: test that folding works
//
int main(int argc, char *argv[]) {
    CGrid grida(RESO,SIZE);
    CGrid gridb(RESO,SIZE);
    CGrid gridc(RESO,SIZE);
    CGrid gridd(RESO,SIZE*2);
    int i;
    int x,y,z;
    double v;
    Vector3D center(RESO*SIZE/2.0,RESO*SIZE/2.0,RESO*SIZE/2.0);
    Vector3D pos;
    Vector3D vx,vy,vz;
    Matrix3D mrot;
    ifstream *ifile;
    ofstream *ofile;
    int cidx;

    // generate a field;
    srand48(time(NULL));

    cerr ≪ "Generating grid" ≪ endl;
    // render a sphere
    for (i=0,x=0;x<grida.getsize();x++) {
        for (y=0;y<grida.getsize();y++) {
            for (z=0;z<grida.getsize();z++,i++) {
                pos=grida.getvector(x,y,z);
                v=Spherefield(RESO,1.0,Vector3D::distance(center,pos));
                grida(i).re=v;
            }
        }
    }

    cerr ≪ "Save and load" ≪ endl;
    ofile=new ofstream(argv[1]);
    (*ofile)≪grida;
    ofile→close();
    delete ofile;

    ifile=new ifstream(argv[1]);
    (*ifile)≫gridb;
    ifile→close();
    delete ifile;

    cerr ≪ "Compare" ≪ endl;
    for (i=0;i<gridb.getsizecubed();i++){
        if (grida(i).im≠gridb(i).im || grida(i).re≠gridb(i).re) {
            cerr≪"A and B differ!"≪i≪":"≪grida(i)≪"!="≪gridb(i)≪endl;
            break;
        }
    }

    // make delta functions, should copy the spehers 2 places
    gridb.clear(0.0,0.0);
    gridb(gridb.getidx(20,20,20)).re=1.0;
    gridb(gridb.getidx(42,42,42)).re=1.0;

    // fold
    cerr ≪ "Fold" ≪ endl;
    gridb.fft();
    grida.fft();
    for (i=0;i<gridb.getsizecubed();i++)
      gridc(i)=grida(i)*gridb(i);
    gridc.ifft();
    gridc.scaledown();

    // save result
    cerr ≪ "Save folded" ≪ endl;
    ofile=new ofstream(argv[1]);
    (*ofile)≪gridc;
    ofile→close();
    delete ofile;

    // test assumptions;
```

```
    cidx=gridc.getidx(gridc.getsizehalf(),
                      gridc.getsizehalf(),
                      gridc.getsizehalf());
    cerr<<"center idx:"<<cidx<<endl;

    // set up base vectors
    cerr<<"Rotate"<<endl;
    vx.set(1.0,0.0,0.0);
    vy.set(0.0,1.0,0.0);
    vz.set(0.0,0.0,1.0);
    mrot.rotation(PI/4.0,0.0,0.0);
//  mrot.rotation(0.0,0.0,0.0);
    vx=vx*mrot;
    vy=vy*mrot;
    vz=vz*mrot;

    // merge rotated grid.
    pos.set(0.0,0.0,gridd.getscale()*gridd.getsizehalf()/2.0);
//  pos.set(0.0,0.0,0.0);
    gridd.clear(0.0,0.0);
    gridc.setscale(0.025);
    gridd.transformfield(gridc,pos,vx,vy,vz);
//  gridd.transformdelta(gridc,pos,vx,vy,vz);

    // save result
    cerr << "Save rotated" << endl;
    ofile=new ofstream(argv[2]);
    (*ofile)<<gridd;
    ofile->close();
    delete ofile;

    return 0;
}
```

testrenderpdb.cc

```
// ************************************************
// File: testrenderpdb.cc
// purpose: test ability to sample density and
// field grids of a given structure with
// different sampling methods
// author: Hans-Henrik Staerfeldt
// notes:

#include "grid.hh"
#include "field.hh"
#include "structure.hh"
#include "vector3d.hh"
#include "stree.hh"
#include <pdb++.h>
#include <sys/types.h>
#include <time.h>
#include <list>
#include <sys/time.h>

using namespace std;

#define ROTATEEULER
#define SIZE 128
#define SCALE 1.0

void usage(void) {
#ifdef ROTATEEULER
    fprintf(stderr,"Usage:  test phi,theta,psi\n");
#else
    fprintf(stderr,"Usage:  test x,y,z,theta\n");
#endif
    exit(1);
}

double randrange(double from, double to){
    return from+drand48()*(to-from);
}

int main( int argc, char *argv[]) {
    ReceptorGridSet gridset1(SCALE,SIZE);
    ReceptorGridSet gridset2(SCALE,SIZE);
    ReceptorGridSet gridset3(SCALE,SIZE);
    struct timeval t_begin;
    struct timeval t_end;
    Vector3D streemin(0.0,0.0,0.0);
    Vector3D streemax(SIZE*SCALE,SIZE*SCALE,SIZE*SCALE);
    Stree <Atom> mystree(streemin,streemax);
    set <Atom *,PtrCmp> result;
    set <Atom *,PtrCmp> atoms;
    set <Atom *,PtrCmp>::iterator j;
    double scale;
    double r;
    Protein prot;
    Vector3D bb1;
    Vector3D bb2;
    Vector3D center;
    Vector3D extend;
    Vector3D zero(0.0,0.0,0.0);
    ofstream of;

    Matrix4D mat;

    try {
        srand48(42);

#ifdef ROTATEEULER
        // euler rotation
        double phi=0.0,theta=0.0,psi=0.0;

        if (argc!=4) {
            theta=randrange(0,2*PI);
            phi=randrange(0,2*PI);
            psi=randrange(0,2*PI);
            cerr<<"Rotation; phi="<<phi;
            cerr<<" theta="<<theta;
            cerr<<" psi="<<psi<<endl;
        }else{
            sscanf(argv[1],"%lf",&phi);
            sscanf(argv[2],"%lf",&theta);
            sscanf(argv[3],"%lf",&psi);
        }
        mat.rotation(phi,theta,psi); // wow make a rotation matrix
#else

        // quaternion rotation
        Vector axis(3);
        double theta=0.0;
        Quaternion q;

        if (argc!=5) {
            axis.set(randrange(0,1),randrange(0,1),randrange(0,1));
            theta=randrange(0,2*PI);
            axis.normalize();
            cerr<<"Rotation; v="<<axis[0]<<",";
            cerr<<axis[1]<<","<<axis[2]<<" theta=";
            cerr<<theta<<endl;
        }else{
            sscanf(argv[3],"%lf",&axis[0]);
            sscanf(argv[2],"%lf",&axis[1]);
            sscanf(argv[1],"%lf",&axis[2]);
            sscanf(argv[1],"%lf",&theta);
        }
        q.set(axis,theta);
        mat=rotation(q);

#endif

        // read protein
        cin >> prot;

        // try rotation
        prot.transform(mat);

        // center entry to fit grid
        prot.getbbox(bb1,bb2);
        cerr<<"bbmin:"<<bb1<<"bbmax:"<<bb2<<endl;

        center=prot.getcog();
        cerr<<"cog:"<<center<<endl;

        r=prot.getdiameter(center);
        cerr<<"radius:"<<r<<endl;

        scale=r*2.0/SIZE;
        cerr<<"minimal grid scale:  "<<scale<<endl;
        cerr<<"real grid scale:  "<<SCALE<<endl;

        // center to rendering space
        center= -center + ((SIZE*SCALE)/2.0);

        cerr << "Translate:  "<<center<<endl;

        prot.translate(center);

        // testing spacetree search
//      cerr<<"build hydrogens"<<endl;
        prot.buildhydrogen();

//      cerr<<"build stree"<<endl;
//      prot.collectatoms(mystree);
//      mystree.print(cerr);
//      prot.collectatoms(atoms);
//      cerr<<"build result"<<endl;
```

```
// zero=streemax/2.0;
// cerr≪"query "≪zero≪","≪8.0≪endl;
// result=mystree.Query(zero,8.0); // at center, w. 8aa radius
// cerr≪"query done"≪endl;

// cerr≪"result size; "≪result.size()≪" atoms"≪endl;
// for (j=result.begin();j≠result.end();j++) {
// if (Vector3D::distance((*j)→pos,zero)>8.0)
// cerr≪"Found atom outside sphere";
// cerr≪(*j)→pos≪endl;
// }
// cerr≪"#"≪endl;
// double sum=0;
// for (j=atoms.begin();j≠atoms.end();j++)
// sum+=(*j)→vdwradius;

// cerr≪"avg radius; "≪sum/atoms.size()≪endl;
// result.clear();
// atoms.clear();

        // render field

#define FOO1
//#define FOO2
//#define FOO3
#define FOO4

#ifdef FOO1
        cerr≪"Render atoms.."≪endl;
        gettimeofday(&t_begin);
        prot.render(gridset1);
        gettimeofday(&t_end);
        cerr≪endl;
        cerr≪"begin:  "≪t_begin.tv_sec≪"s "≪t_begin.tv_usec≪endl;
        cerr≪"end:  "≪t_end.tv_sec≪"s "≪t_end.tv_usec≪endl;
        cerr≪"Done in "≪t_end.tv_sec-t_begin.tv_sec≪endl;
#endif


#ifdef FOO2
        cerr≪"Render fast tree.."≪endl;
        gettimeofday(&t_begin);
        prot.rendertreefast(gridset2);
        gettimeofday(&t_end);
        cerr≪endl;
        cerr≪"begin:  "≪t_begin.tv_sec≪"s "≪t_begin.tv_usec≪endl;
        cerr≪"end:  "≪t_end.tv_sec≪"s "≪t_end.tv_usec≪endl;
        cerr≪"Done in "≪t_end.tv_sec-t_begin.tv_sec≪endl;
#endif

#ifdef FOO3
        cerr≪"Render tree.."≪endl;
        gettimeofday(&t_begin);
        prot.rendertree(gridset3);
        gettimeofday(&t_end);
        cerr≪endl;
        cerr≪"begin:  "≪t_begin.tv_sec≪"s "≪t_begin.tv_usec≪endl;
        cerr≪"end:  "≪t_end.tv_sec≪"s "≪t_end.tv_usec≪endl;
        cerr≪"Done in "≪t_end.tv_sec-t_begin.tv_sec≪endl;
#endif

        of.open("data/testrender.ent");
        of≪prot;
        of.close();

#ifdef FOO4
        // output field
        cerr≪"Writing fields.."≪endl;
#ifdef FOO1
        gridset1.save("data/testrender.atom");
#endif
#ifdef FOO2
        gridset2.save("data/testrender.fasttree");
#endif
#ifdef FOO3
        gridset3.save("data/testrender.tree");
#endif

#endif
    }
    catch ( char *e ){
        cerr≪"Caught exception:  "≪e≪endl;
        return -1;
    }
}
```

testrotamer.cc

```
// ************************************************
// File: testrotamer.cc
// purpose: test ability to use rotamer libraries
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "rotamer.hh"
#include "structure.hh"
#include "grid.hh"
#include <fstream.h>
#include <iostream.h>

#define RESOLUTION 0.25
#define SIZE 64

void showrecord(rotamerrecord *rec) {
    cout ≪ rec→ident;
    cout ≪" "≪ rec→phi;
    cout ≪" "≪ rec→psi;
    cout ≪" "≪ rec→N;
    cout ≪" "≪ rec→r[0];
    cout ≪" "≪ rec→r[1];
    cout ≪" "≪ rec→r[2];
    cout ≪" "≪ rec→r[3];
    cout ≪" "≪ rec→p;
    cout ≪" "≪ rec→chi[0];
    cout ≪" "≪ rec→chi[1];
    cout ≪" "≪ rec→chi[2];
    cout ≪" "≪ rec→chi[3];
    cout ≪endl;
}

int main( int argc, char *argv[]) {
    rotamerlibrary myrotlib;
    rotamerrecord rec;
    Protein prot;
    ifstream pdbin;
    ofstream pdbout;
    Matrix4D m;
    double r;
    char filename[64];
    char *strloc[3]={"CT","C","NT"};
    int i,j;
    int placement;

    try {
        myrotlib.open("data/rotamer.lib");
        myrotlib.report();

        for (j=RESIDUE_START;j<RESIDUE_MAX;j++) {

            for (placement=RESIDUE_CTERMINUS;
                  placement≤RESIDUE_NTERMINUS;
                  placement++) {

                // load each type up and assign it a random rotamer
                sprintf(filename,"residues/%s.ent",ResidueInfoData[j].key);
                pdbin.open(filename);

                if (pdbin.good()) {

                    try {
                        prot.setsingleresidueplacement(ResiduePlacementtype(placement));
                        pdbin ≫ prot;                              /* read residue file */
                        // transform protein to have the
                        // N atom at (0,0,0)
                        // CA along the x-axis and
                        // C defining the z-axis
                        m.viewat(prot.residue[0][RATOM_N].pos,
                                 prot.residue[0][RATOM_CA].pos,
                                 prot.residue[0][RATOM_C].pos);

                        prot.transform(m);
                        prot.translate(Vector3D(RESOLUTION*SIZE/2.0,
                                                RESOLUTION*SIZE/2.0,
                                                RESOLUTION*SIZE/2.0));
                        cerr≪"placement:"≪prot.residue[0].getplacement()≪endl;

                        r=prot.getdiameter(prot.residue[0][RATOM_N].pos);
                        cout≪ResidueInfoData[j].key≪" diameter:"≪r≪endl;

                        rec.chi[0]=0.0;
                        rec.chi[1]=0.0;
                        rec.chi[2]=0.0;
                        rec.chi[3]=0.0;
                        rec.phi=0.0;
                        rec.psi=0.0;

                        if (myrotlib.set_type(ResidueInfoData[j].key)==0){
                            // this is in the library
```

```
                cout ≪ ResidueInfoData[j].key ≪ " In library"≪endl;
                myrotlib.get_record(&rec); // gets the first record
             }else
               cout ≪ ResidueInfoData[j].key ≪ " Not In library"≪endl;
             i=0;

             cout ≪ "Angles; ( phi,psi:" ≪ rec.phi ≪ ",";
             cout ≪ rec.psi ≪ " chi[4]:";
             cout ≪ rec.chi[0] ≪ "," ≪ rec.chi[1] ≪ ",";
             cout ≪ rec.chi[2] ≪ "," ≪ rec.chi[3] ≪ ")"≪endl;

             // build residue according to chi values of record
             prot.residue[0].buildchi(rec.phi,rec.psi,rec.chi);

             // save structure
             sprintf(filename,"residues/%s.%d.%s.ent",
                     ResidueInfoData[j].key,i,strloc[placement]);

             pdbout.open(filename);
             pdbout ≪ prot;
             pdbout.close();



          }
          catch (char *error) {
             cerr≪"Caught exception:"≪error≪endl;
          }
          pdbin.close();
        }
      }
    }
  }
  catch (char *error) {
    cerr≪"Caught exception:"≪error≪endl;
    flush(cerr);
  }
  return 0;
}
```

testrotamerfields.cc

```
// *************************************************
// File: testrotamerfields
// purpose: test the ability to read rotamers, build the
// density and field grids and save them for
// all rotamers
// author: Hans-Henrik Staerfeldt
// notes:

#include "rotamer.hh"
#include "structure.hh"
#include "grid.hh"
#include <fstream.h>
#include <iostream.h>
#include <time.h>

#define RESOLUTION 0.3
#define SIZE 64

void usage( void ) {
  cerr ≪ "testrotamerfields <RES>" ≪ endl;
  exit(1);
}

int main( int argc, char *argv[]) {
  rotamerlibrary myrotlib;
  rotamerrecord rec;
  Protein prot;
  ifstream *pdbin;
  Matrix4D m;
  ReceptorGridSet mygrid(RESOLUTION,SIZE);
  ReceptorGridSet mygridc(RESOLUTION,SIZE);

  double r;
  int recordnum;
  char *filename=argv[1];

  if (argc≠2) usage();

  try {

    myrotlib.open("data/rotamer.lib");
    //myrotlib.report();

    // load each type up and assign it a random rotamer
    pdbin = new ifstream(filename);
    *pdbin ≫ prot;                                   /* read residue file */
    delete pdbin;

    // transform protein to have the
    // N atom at (0,0,0)
    // CA along the x-axis and
    // C defining the z-axis
    m.viewat(prot.residue[0][RATOM_N].pos,
             prot.residue[0][RATOM_CA].pos,
             prot.residue[0][RATOM_C].pos);

    prot.transform(m);
    prot.translate(Vector3D(RESOLUTION*SIZE/2.0,
                            RESOLUTION*SIZE/2.0,
                            RESOLUTION*SIZE/2.0));

    r=prot.getdiameter(prot.residue[0][RATOM_N].pos);

    cerr≪filename≪" diameter:"≪r≪"AA"≪endl;
    cerr≪filename≪" gridsize:"≪SIZE≪"^3 ";
    cerr≪RESOLUTION*SIZE≪"AA"≪endl;
    cerr≪filename≪" resolution:"≪RESOLUTION≪"AA/cell"≪endl;

    myrotlib.set_type(ResidueInfoData[prot.residue[0].type].key);

    // just test random record

    srand48(time(NULL));
    srand48(42);
    recordnum = drand48()*myrotlib.get_count();

    cerr ≪ filename ≪ " rotamer record number :";
    cerr ≪ recordnum ≪"/"≪ myrotlib.get_count()≪ endl;
    myrotlib.set_index(recordnum);

    myrotlib.get_record(&rec);

    // build residue according to chi values of record
    prot.residue[0].buildchi(0.0,0.0,rec.chi);

    // render the fields of this residue
    mygrid.clear();
    prot.render(mygrid);

    // save field
    cout ≪ mygrid.ShapeR;

    return 0;
  }
  catch ( char *e ){
    cerr≪"Caught exception:  "≪e≪endl;
    return -1;
  }
}
```

testrotatepdb.cc

```
// *************************************************
// File: testrotatepdb
// purpose: test ability to read, rotate and write a pdb file
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "structure.hh"
#include "vector3d.hh"
#include <pdb++.h>

//#define ROTATEEULER

void usage(void) {
#ifdef ROTATEEULER
  fprintf(stderr,"Usage:  rotatepdb phi theta psi "
          "< pdb.ent > pdb.rot.ent \n");
#else
  fprintf(stderr,"Usage:  rotatepdb x y z theta "
          "< pdb.ent > pdb.rot.ent \n");
#endif
  exit(1);
}

int main( int argc, char *argv[]) {
  Protein prot;
  Matrix4D mat;

  try {
#ifdef ROTATEEULER
    // euler rotation
    double phi=0.0,theta=0.0,psi=0.0;
```

```
        if (argc!=4) usage();

        sscanf(argv[1],"%lf",&phi);
        sscanf(argv[2],"%lf",&theta);
        sscanf(argv[3],"%lf",&psi);

        mat.rotation(phi,theta,psi); // wow make a rotation matrix
#else

        // quaternion rotation
        Vector3D axis;
        double theta=0.0;
        Quaternion q;

        if (argc!=5) usage();

        sscanf(argv[3],"%lf",&axis[0]);
        sscanf(argv[2],"%lf",&axis[1]);
        sscanf(argv[1],"%lf",&axis[2]);
        sscanf(argv[1],"%lf",&theta);

        q.set(axis,theta);
        mat.rotation(q);

#endif
        cerr<<"reading"<<endl;
        cin >> prot; // oooh read it.
        cerr<<"rotating"<<endl;
        prot.transform(mat); // coool, rotate it
        cerr<<"building"<<endl;
        prot.buildhydrogen();
        cout << prot; // hello! print it
    }
    catch ( char *e ){
        cerr<<"Caught exception:  "<<e<<endl;
        return -1;
    }

}
```

testrotationspace.cc

```
// **************************************************
// File: rotationspace.cc
// purpose: test rotationspace
//
// author: Hans-Henrik Staerfeldt
// notes:
// generates a pdb file with atoms at the
// surface of the rotational subdivision sphere

#include "rotationspace.hh"
#include "vector3d.hh"
#include <pdb++.h>

void show( double phi, double theta, double psi, char *atm) {
    static int i=1;
    Vector3D pos;
    PDB rec;

    rec.type(PDB::HETATM);
    rec.hetatm.serialNum=i+1;
    strcpy(rec.hetatm.name,atm);
    rec.hetatm.altLoc=' ';
    strcpy(rec.hetatm.residue.name,"HOH");
    rec.hetatm.residue.chainId=' ';
    rec.hetatm.residue.seqNum=0;
    rec.hetatm.residue.insertCode=' ';
    pos.setspherical(phi,theta,(PI2+psi)*1.2);
    rec.hetatm.xyz[0]=pos.x();
    rec.hetatm.xyz[1]=pos.y();
    rec.hetatm.xyz[2]=pos.z();
    rec.hetatm.occupancy=0;
    rec.hetatm.tempFactor=0;
    rec.hetatm.ftnoteNum=0;
    cout<<rec<<endl;
    i++;
}

void main (int argc, char *argv[]) {

    Rotationspace <int> rot;
    vector <Vector3D> leafs;
    double phi;
    double theta;
    double psi;
    Matrix4D mat;
    int i;

    try {

        phi=-0.493623;
        theta=0;
        psi=0.261799;
        Rotationspace::getmatrix(phi,theta,psi,mat);

        phi=0;
        theta=-0.493623;
        psi=0.261799;
        Rotationspace::getmatrix(phi,theta,psi,mat);

        phi=0.261799;
        theta=-0.493623;
        psi=0;
        Rotationspace::getmatrix(phi,theta,psi,mat);

        cerr<<rot.size()<<" angles"<<endl;

        // You can also subdivide evenly deep at all places.

// cerr<<rot.size()<<" angles"<<endl;

        rot.collectsegments(leafs);

        cerr<<leafs.size()<<" angles"<<endl;

        // write for visualization
        for (i=0;i<leafs.size();i++){
            if (leafs[i].z()==0.0)
                show(leafs[i].x(),leafs[i].y(),leafs[i].z(),"O");
        }

        rot.subdivide(3);
        leafs.clear();
        rot.collectleafs(leafs);
        cerr<<leafs.size()<<" angles"<<endl;

        // write for visualization
        for (i=0;i<leafs.size();i++){
            if (leafs[i].z()<0.3)
                show(leafs[i].x(),leafs[i].y(),leafs[i].z(),"C");
        }
// rot.render();
    }
    catch ( char *error ){
        cerr<<"Caught exception "<<error<<endl;
    }
}
```

teststree.cc

```
// **************************************************
// File: testree.cc
// purpose: test space partitioning tree
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "stree.hh"
#include "vector3d.hh"
#include <stdlib.h>

class myData {
 public:
    StreeElement <myData> *element;
    int type;
    double x;
    double y;
    double z;
};

myData data[1000];

int main(int argc, char *argv[]) {
    Vector3D min(0,0,0);
    Vector3D max(100,100,100);
    Stree <myData> myStree(min,max);
    Vector3D p;
    int i;

    try {
        for (i=0;i<1000;i++){
            p.set(drand48()*100,drand48()*100,drand48()*100);
            data[i].x=p[0];
            data[i].y=p[1];
            data[i].z=p[2];
```

```
        data[i].element=myStree.Add(p,data[i]);
      }

      for (i=0;i<1000;i++){
        p.set(drand48()*100,drand48()*100,drand48()*100);
        data[i].x=p[0];
        data[i].y=p[1];
        data[i].z=p[2];
        data[i].element→Update(p);
      }

      return 0;
   }
   catch ( char *e ){
      cerr≪"Caught exception:  "≪e≪endl;
      return -1;
   }

}
```

testsurfacepdb.cc

```
// **************************************************
// File: testsurfacepdb.cc
// purpose: test surface measure
//
// author: Hans-Henrik Staerfeldt
// notes:
// assumes that surface is set to be written in
// structure.cc

#include "structure.hh"
#include "vector3d.hh"
#include <pdb++.h>

void usage(void) {
   fprintf(stderr,"Usage:  testsurfacepdb < pdb.ent > pdb.out.ent \n");
   exit(1);
}

int main( int argc, char *argv[]) {
   Protein prot;
   Vector3D cog;

   if (argc≠1) usage();

   try {
      cerr≪"reading"≪endl;
      cin ≫ prot; // oooh read it.
      cog=prot.getcog();
      prot.translate(-cog); // center it.
      prot.buildhydrogen();
      cerr≪"finding surface"≪endl;
      prot.findsurface(1.7); // water
      cerr≪"writing"≪endl;
      cout ≪ prot; // hello! print it
   }
   catch ( char *e ){
      cerr≪"Caught exception:  "≪e≪endl;
      return -1;
   }

}
```

testvector.cc

```
// **************************************************
// File: testvector.cc
// purpose: test vector arithmetic
//
// author: Hans-Henrik Staerfeldt
// notes:

#include "vector3d.hh"
#include <math.h>

int main( int argc, char *argv[]){
   Vector3D v1(1.0,2.0,0.3);
   Vector3D v2(1.2,0.2,2.3);
   Vector4D v3;
   Matrix3D m1;
   Matrix4D m2;

   try {
      cout ≪ v1 ≪ "+" ≪ v2 ≪ "=" ≪ v1+v2 ≪endl;
      // v3=v2;
      cout ≪ v3 ≪ "?" ≪ v2 ≪endl;
      cout ≪ m2;

      cout ≪ "acos(-1)="≪acos(-1)≪endl;
      cout ≪ "acos(-0.5)"≪acos(-0.5)≪endl;
      cout ≪ "acos(0)"≪acos(0)≪endl;
      cout ≪ "acos(0.5)"≪acos(0.5)≪endl;
      cout ≪ "acos(1)"≪acos(1)≪endl;

      cout ≪ "asin(-1)"≪asin(-1)≪endl;
      cout ≪ "asin(-0.5)"≪asin(-0.5)≪endl;
      cout ≪ "asin(0)"≪asin(0)≪endl;
      cout ≪ "asin(0.5)"≪asin(0.5)≪endl;
      cout ≪ "asin(1)"≪asin(1)≪endl;

     return 0;
   }
   catch ( char *e ){
      cerr≪"Caught exception:  "≪e≪endl;
      return -1;
   }

}
```