# New approaches to protein docking

Dissertation zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

von

Oliver Kohlbacher

Saarbrücken
12. Januar 2001

Datum des Kolloquiums: 12. Januar 2000

Dekan der technischen Fakultät:
Professor Dr. Rainer Schulze-Pillot-Ziemen

Gutachter:
Professor Dr. Hans-Peter Lenhof, Universität des Saarlandes, Saarbrücken
Professor Dr. Kurt Mehlhorn, MPI für Informatik, Saarbrücken

"I think the most exciting computer research now is partly in robotics, and partly in applications to biochemistry.

. . .

Biology is so digital, and incredibly complicated, but incredibly useful. The trouble with biology is that, if you have to work as a biologist, it's boring. Your experiments take you three years and then, one night, the electricity goes off and all the things die! You start over. In computers we can create our own worlds. Biologists deserve a lot of credit for being able to slug it through." – *Donald Knuth*

# Acknowledgements

The work on this thesis was carried out during the years 1996–2000 at the Max-Planck-Institut für Informatik in the group of Prof. Dr. Kurt Mehlhorn under the supervision of Prof. Dr. Hans-Peter Lenhof.

Prof. Dr. Hans-Peter Lenhof kindled my interest in Bioinformatics and gave me the freedom to do research in those areas that fascinated me most. Our discussions, although sometimes heated, were always fruitful and forced me to get to the very bottom of many problems.

The implementation of BALL is unthinkable without the help of all the people who contributed code and ideas. Nicolas Boghossian had significant impact on the design of the library core and contributed lots of code for the kernel. Heiko Klein implemented the largest part of the visualization. Dr. Peter Müller brought in his experience in the field of molecular dynamics simulations. Andreas Burchardt implemented parts of the NMR code. Upon Andreas Moll fell the ungrateful work of testing and debugging. Stefan Strobel implemented the difficult part of molecular surface calculations. Andreas Hildebrandt implemented the NMR visualization. The more sophisticated parts of the solvation code were implemented by Andreas Kerzmann, who also set up the BALL web server. Last, but not least, Prof. Dr. Hans-Peter Lenhof not only contributed code for structure mapping and force field calculations, but also initiated the whole project and kept everyone motivated.

Ernst Althaus implemented the branch-&-cut-algorithm and was coauthor of the paper on flexible docking. He had the patience to explain to me some of the finer details of polyhedral theory.

The Rechnerbetriebsgruppe had to suffer from this work as well. Jörg Herrmann, Wolfram Wagner, Bernd Färber, Uwe Brahm, Thomas Hirtz, and Roland Berberich solved numerous of my hardware- and software-related problems even at night and during weekends. Christoph Clodo managed to track down and resolve several errors in the LaTeX code of this thesis.

I am also grateful to my “WG” (Holger, Michael, Christian) for nearly six years of enjoyable coexistence and for the flat, the evenings, and quite some bottles of wine we shared. Last, but certainly not least, I wish to thank Andreas for his patience, his understanding, and his support while I wrote this thesis.

# Abstract

In the first part of this work, we propose new methods for protein docking. First, we present two approaches to protein docking with flexible side chains. The first approach is a fast greedy heuristic, while the second is a branch-&-cut algorithm that yields optimal solutions. For a test set of protease-inhibitor complexes, both approaches correctly predict the true complex structure. Another problem in protein docking is the prediction of the binding free energy, which is the the final step of many protein docking algorithms. Therefore, we propose a new approach that avoids the expensive and difficult calculation of the binding free energy and, instead, employs a scoring function that is based on the similarity of the proton nuclear magnetic resonance spectra of the tentative complexes with the experimental spectrum. Using this method, we could even predict the structure of a very difficult protein-peptide complex that could not be solved using any energy-based scoring functions.

The second part of this work presents BALL (Biochemical ALgorithms Library), a framework for Rapid Application Development in the field of Molecular Modeling. BALL provides an extensive set of data structures as well as classes for Molecular Mechanics, advanced solvation methods, comparison and analysis of protein structures, file import/export, NMR shift prediction, and visualization. BALL has been carefully designed to be robust, easy to use, and open to extensions. Especially its extensibility, which results from an object-oriented and generic programming approach, distinguishes it from other software packages.



# Kurzzusammenfassung

Der erste Teil dieser Arbeit beschäftigt sich mit neuen Ansätzen zum Proteindocking. Zunächst stellen wir zwei Ansätze zum Proteindocking mit flexiblen Seitenketten vor. Der erste Ansatz beruht auf einer schnellen, gierigen Heuristik, während der zweite Ansatz auf branch-&-cut-Techniken beruht und das Problem optimal lösen kann. Beide Ansätze sind in der Lage die korrekte Komplexstruktur für einen Satz von Testbeispielen (bestehend aus Protease-Inhibitor-Komplexen) vorherzusagen. Ein weiteres, grösstenteils ungelöstes, Problem ist der letzte Schritt vieler Protein-Docking-Algorithmen, die Vorhersage der freien Bindungsenthalpie. Daher schlagen wir eine neue Methode vor, die die schwierige und aufwändige Berechnung der freien Bindungsenthalpie vermeidet. Statt dessen wird eine Bewertungsfunktion eingesetzt, die auf der Ähnlichkeit der Protonen-Kernresonanzspektren der potentiellen Komplexstrukturen mit dem experimentellen Spektrum beruht. Mit dieser Methode konnten wir sogar die korrekte Struktur eines Protein-Peptid-Komplexes vorhersagen, an dessen Vorhersage energiebasierte Bewertungsfunktionen scheitern.

Der zweite Teil der Arbeit stellt BALL (Biochemical ALgorithms Library) vor, ein Rahmenwerk zur schnellen Anwendungsentwicklung im Bereich Molecular Modeling. BALL stellt eine Vielzahl von Datenstrukturen und Algorithmen für die Felder Molekülmechanik, Vergleich und Analyse von Proteinstrukturen, Datei-Import und -Export, NMR-Shiftvorhersage und Visualisierung zur Verfügung. Beim Entwurf von BALL wurde auf Robustheit, einfache Benutzbarkeit und Erweiterbarkeit Wert gelegt. Von existierenden Software-Paketen hebt es sich vor allem durch seine Erweiterbarkeit ab, die auf der konsequenten Anwendung von objektorientierter und generischer Programmierung beruht.

# Contents

**Part I – Introduction** **1**

**Part II – Protein Docking** **7**

**1 Biochemistry – the Basics** **9**
1.1 Atoms and Molecules . . . 9
1.2 Amino Acids . . . 10
1.3 Proteins . . . 10
1.4 Nucleic Acids . . . 13
1.5 Interatomic Forces . . . 14
1.5.1 Nonbonded interactions . . . 14
1.5.2 Molecular Mechanics . . . 16

**2 Introduction** **19**
2.1 Rigid Body Docking . . . 19
2.2 Docking and Protein Flexibility . . . 21
2.3 Combining NMR Data and Docking Algorithms . . . 23

**3 Semi-Flexible Docking** **25**
3.1 Introduction . . . 25
3.2 The Docking Algorithm . . . 28
3.2.1 Rigid Docking . . . 28
3.2.2 Side Chain Demangling . . . 28
3.2.3 The Multi-Greedy Method . . . 29
3.2.4 The Branch-&-Cut Algorithm . . . 30
3.2.5 Energetic Evaluation . . . 38
3.3 Experimental Results . . . 40

**4 Protein Docking and NMR** **45**
4.1 Nuclear Magnetic Resonance Spectroscopy . . . 45
4.1.1 The Nuclear Angular Momentum . . . 45
4.1.2 Electronic Shielding and the Chemical Shift . . . 47
4.1.3 The Basic NMR Experiment . . . 48
4.2 Application to the Protein Docking Problem . . . 50
4.2.1 Previous Work . . . 50
4.2.2 NMR Shift Prediction . . . 50
4.2.3 Spectrum Synthesis and Comparison . . . 53

4.3 Experimental Results . . . 55
4.3.1 Methods . . . 55
4.3.1.1 Preparation of Structures and Rigid Body Docking . . . 55
4.3.1.2 NMR Chemical Shift Calculation . . . 55
4.3.1.3 Spectrum Synthesis and Comparison . . . 56
4.3.2 Results . . . 57

**5 Discussion** **61**

**Part III – BALL** **65**

**6 Design and Implementation** **67**
6.1 Introduction . . . 67
6.2 Design Goals . . . 68
6.2.1 Ease of Use . . . 68
6.2.2 Functionality . . . 69
6.2.3 Openness . . . 69
6.2.4 Robustness . . . 69
6.3 Choice of Programming Language . . . 70
6.4 Architecture . . . 70
6.5 The Foundation Classes . . . 71
6.5.1 Global Definitions . . . 71
6.5.2 Composite Class . . . 72
6.5.3 Object Persistence . . . 73
6.5.4 Run-Time Type Identification . . . 78
6.5.5 Iterators . . . 78
6.5.6 Processors . . . 79
6.5.7 Options . . . 80
6.5.8 Logging Facility . . . 81
6.5.9 Strings and Related Classes . . . 82
6.5.10 Mathematics . . . 82
6.5.11 Miscellaneous . . . 82
6.6 The Kernel . . . 83
6.6.1 Molecular Data Structures . . . 83
6.6.2 Iterators . . . 85
6.6.3 Selection . . . 85
6.7 The Basic Components . . . 87
6.7.1 File Import/Export . . . 87
6.7.2 Molecular Mechanics . . . 87
6.7.3 Nuclear Magnetic Resonance Spectroscopy . . . 95
6.7.4 Visualization . . . 96
6.8 Scripting Language Integration . . . 101
6.8.1 Python . . . 101

6.8.2 Extending . . . 102
6.8.3 Embedding . . . 104

**7 Project Management** **107**
7.1 Revision Management . . . 107
7.2 Coding Conventions and Software Metrics . . . 107
7.3 Portability . . . 108
7.4 Documentation . . . 108
7.4.1 Reference Manual and Tutorial . . . 108
7.4.2 FAQs . . . 109
7.5 Testing . . . 109
7.5.1 Fundamentals . . . 111
7.5.2 Testing in BALL . . . 111
7.5.3 Test macros . . . 112
7.5.4 Automatic Test Builds . . . 115
7.6 Installation and Configuration . . . 116

**8 Programming with BALL** **119**

**9 Outlook** **123**

**Part IV – Conclusion** **125**

**A UML Notation** **129**

**Appendix** **129**

**B Curriculum Vitae** **131**

**C Bibliography** **134**

**Index** **141**

# Part I

# Introduction

## Motivation

At the beginning of the 21st century, biology has emerged as the leading science. It is mainly driven by the increasing economic impact of molecular genetics, biochemistry, medicine, and pharmaceutics. These disciplines form the cornerstones of a new scientific field for which the term *Life Science* was coined.

In the last few years, the most important discoveries in Life Science were made in the field of *genomics*. Starting with the first completely sequenced genome of a living organism (the genome of bacterium *Haemophilus influenzae* [35]) in 1995, genomics rapidly developed: the world's sequencing capacity has not yet stopped its exponential growth and complete genomes of microbial organisms are currently being sequenced routinely. The complete sequence of the human genome represents an important milestone for genomics – the data emerging from this project will be the basis of molecular medicine for decades. The completion of this project will also change the focus of genomics. Initially, genomics focused on the acquisition of genomes. With the availability of this data, genomics will have to attack the next question: What is the meaning of all this data?

The exploration of gene activity and regulation is one central issue in this context. The second big issue is summarized with just another catchword: *proteomics* – the study of proteins, their function and expression. Genomics and proteomics are obviously tightly interwoven and a comprehensive understanding of the molecular basis of life is to be expected from their close interaction.

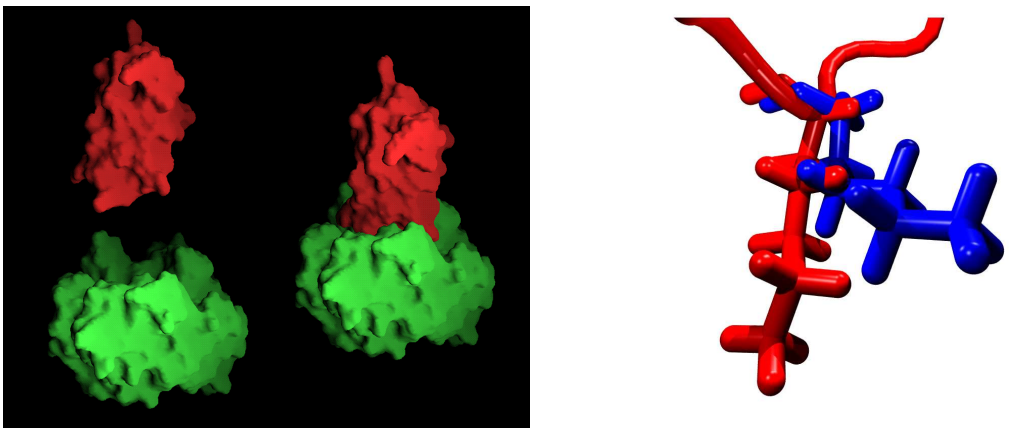
The most important economic driving force behind these developments is the pharmaceutical industry, which hopes to profit from a deeper understanding of molecular medicine for the development of new drugs. Thus, the ultimate goal is true rational drug design, which means designing a drug based on a thorough understanding of its molecular targets and their interactions.

## Protein Docking

The theoretical prediction of these interactions is of prime importance since it permits the verification of hypotheses in the course of drug development without expensive and time-consuming "wet" experiments. One of problems arising in drug design is the so-called *protein-protein docking problem*: Given the structures of the proteins $A$ and $B$ that are known to form a complex $AB$, predict the structure of this complex.

In 1894, German chemist Emil Fischer proposed the so-called *lock-and-key principle* [34]. It states, that the selective binding of two proteins is caused by geometric complementarity. $A$ and $B$ each possesses a characteristic shape (like a lock and its key). The two proteins can only aggregate if they share complementary regions, *i.e.* if $A$ "fits" into $B$.

The first algorithms for protein docking were strictly based on geometric complementarity. They also assumed that both proteins were rigid bodies and did not change their structures on binding. For most standard examples, this assumption is reasonable, so there are a number of protein docking algorithms that use rigid docking approaches. However, there are some prominent examples where one or both proteins undergo structural changes upon binding and thus for which these approaches fail. We will briefly illustrate the problem with the example of



(a) (b)

Figure 1: **(a)** *The enzyme trypsin (green) and its inhibitor BPTI (red) form a complex (right side).* **(b)** *Several amino acids of BPTI change their structure on binding. This figure shows the structure of one of these amino acids (LYS:15) in its native structure (red) and in the complex structure (blue). The structural change in this amino acid impedes the prediction of the complex structure using rigid docking approaches.*

trypsin (an enzyme) and BPTI (the bovine pancreatic trypsin inhibitor), one of the best studied protein complexes.

The unbound structures (*native structure*) of trypsin and BPTI as well as the structure of their complex have been known for a long time. The larger trypsin (see Fig. 1(a)) binds the smaller BPTI tightly. When subjected to a rigid docking prediction, most algorithms fail inevitably. The reason can be seen in Fig. 1(b): the conformation of several amino acids at the tip of BPTI in the native structure differs considerably from their conformation in the complex. Hence, rigid-body docking algorithms have little chance of predicting the correct structure – there is no geometric complementarity in the native structures. Thus, to solve that class of protein docking problems, we have to consider at least the side chains to be flexible, while the backbone still remains rigid. This is what we call *semi-flexible docking*. We have developed two new approaches for semi-flexible protein docking. Both approaches are based on a rigid-body docking algorithm, which produces a number of tentative complex structures. In the next step, we optimize the side chains in the binding site of the tentative complex. Finding the optimal side chain conformation is a very difficult (in the order of 200–500 dimensional) continuous optimization problem. By restricting the conformational space of the side chains to a set of preferred conformations (so-called rotamers), we can reduce the problem to a combinatorial optimization problem. The optimization problem is still rather difficult: identify the set of rotamers with minimal energy out of $\approx 10^{60}$ possible combinations. For this side chain placement step, we propose two new algorithms. The first approach is a branch-&-cut algorithm based on the ILP (integer linear programming) formulation of the problem. We are able to identify some classes of facet-defining inequalities and we devised a separation algorithm

for a subclass of inequalities. The second approach is a fast, simple greedy approach. In contrast to the branch-&-cut algorithm, the greedy approach yields usually suboptimal solutions. Nevertheless, both approaches were able to demangle the side chains of a test set of protease-inhibitor complexes. A final energetic evaluation of the demangled conformations correctly predicts the true complex structure. After a short introduction to protein docking in general (Chapter 2), we describe these approaches for semi-flexible docking in detail in Chapter 3.

Another major problem in protein docking is the accurate prediction of the binding free energy of tentative complex structures, which is usually the basis for the final ranking performed by the docking algorithm. Although considerable progress has been made with the prediction of the binding free energy, it remains largely an unsolved problem. We develop a novel scoring function for protein-protein docking that is based on the integration of experimental data. This scoring function ranks the tentative complex structures with respect to the deviation of the predicted nuclear magnetic resonance spectrum (NMR spectrum) from the experimental spectrum. Since NMR spectra (especially the proton-NMR spectra we use in this work) are easily accessible for protein complexes, the use of experimental data can improve the quality of docking predictions significantly and can be used to assess the reliability of the results. We develop techniques for the prediction of NMR chemical shifts, the reconstruction of NMR spectra from the shift data, and for spectra comparison. This novel approach is the first docking approach that permits the direct integration of experimental data into the docking algorithm. Its application to a set of protein-protein and protein-peptide complexes gives very promising results. Using this new technique, we can also solve a very difficult example of a protein-peptide complex, where all energy-based scoring functions fail. The methods and techniques developed for NMR-based docking are described in depth in Chapter 4 after a short general introduction to NMR spectroscopy.

### Rapid Software Prototyping

While developing new methods in protein docking as well as in other areas of Computational Molecular Biology, the most time-consuming step is the implementation. A major portion of this time is spent on the implementation of fundamental data structures and algorithms. The data structures and algorithms are usually reimplemented again and again; code reuse is not very common. We illustrate the main reasons for this with an example. In the course of our protein docking project, we experimented with advanced methods for energetic evaluations. One of these methods [54] consists of two major calculation steps: the calculation of the electrostatic contributions and the calculation of the molecular surface. Both methods are standard techniques that were developed about 15 years ago and several implementations exist. We then bought a commercially available software package for the electrostatic calculations and chose one of the freely available implementations for the molecular surface area calculation. Both programs were written in FORTRAN 77, the most common language for this kind of software. The integration of subroutines or major code portions from these two implementations into a common program proved to be nearly impossible, since no thought was spent on reusability when designing the software. Furthermore, the lack of documentation or even comments in conjunction with FORTRAN-specific coding habits (*e.g.*, one- or two-letter variable names) tends to turn even minor changes into a nightmare.

The only means of integration we found was to implement rather large amounts of so-called “glue code”. This code had to convert the input data to the specific file formats required by the two different programs. Although both programs basically used the PDB format to read atom coordinates, they required two different variants of the format which were incompatible to each other. In the end, we implemented several hundred lines of code just to make both programs read standard PDB files. The extraction of the results from the (text-based) output of the programs again required rather complex code to gather all relevant numbers and ensure that the code really had correctly terminated. The resulting collection of interacting C programs, FORTRAN programs, and shell scripts was even less maintainable and less comprehensible than the original FORTRAN code.

We then looked at several large molecular modeling packages. The producers of these packages usually promise a good integration of several standard components. Often, there are also software development kits (SDKs) available for the extension of existing methods and the integration of new methods. But even these incredibly expensive packages have major drawbacks. Some of these packages are just nice graphical front ends to the very same FORTRAN packages mentioned above; they are basically glue code with windows. The SDKs have exclusively procedural interfaces. We could not find any object-oriented approaches, which were preferable with respect to reusability and extensibility of the code. Besides SDKs, many packages also provide scripting languages that can be used to implement additional functionality. However, each package defines its own cryptic language, which often enough lacks even basic control structures, so these approaches are seldom satisfactory.

There are also some (academic) efforts to create class libraries and frameworks in the field of Computational Molecular Modeling. Some prominent examples are PDBLib [18] (a library for processing PDB files) and SCL [118]. These libraries have some promising features, but none of the existing libraries could provide us with sufficiently broad functionality.

Finally, there is is small number of scripting language extensions for Molecular Modeling and Computational Molecular Biology. *BioPerl* [12] and *BioPython* [13] are two projects that started rather recently and do not yet provide any functionality in the field of Molecular Modeling. The software package that came closest to meeting our needs is *MMTK*, the *Molecular Modeling Toolkit*. MMTK is an extension package for the object-oriented scripting language Python [121] and provides some basic functionality for Molecular Modeling and visualization. Due to its object-oriented concept, MMTK is open and extensible on the scripting language level. The time-critical sections of the code are implemented in C, resulting in good performance, but hindering reuse. Hence, the main drawbacks of MMTK are its limited functionality and extensibility. Nevertheless, this package demonstrates the advantages of an object-oriented scripting language.

Our experience with these software packages led us to recognize the need for an extensible, efficient, and object-oriented tool kit in the field of Molecular Modeling, which we then started to implement. During the four years of its implementation, this tool set quickly evolved into something larger: it became a framework for rapid application development in Molecular Modeling – BALL, the Biochemical Algorithms Library.

BALL is a large and powerful framework for rapid software prototyping written in C++. We use state-of-the-art software engineering techniques to ensure a thorough design and high code quality. The resulting code is portable, robust, efficient, and extensible. Especially its

extensibility, which results from an object-oriented and generic programming approach, distinguishes BALL from all other software packages.

BALL provides an extensive set of data structures as well as classes for molecular mechanics, advanced solvation methods, comparison and analysis of protein structures, file import/export, and visualization. To reduce turn-around times while developing new methods, we added Python bindings for the majority of the BALL classes. Using the scripting language, it is easier to inspect the data structures interactively. Additionally, the code can be modified at run time without time-consuming compile or link stages. Once the Python code works as expected, it is very simple to port the Python code to C++. We have proven the rapid software prototyping capabilities of BALL for a number of example applications in the field of protein docking and protein engineering. The new techniques for protein docking described in the first part of this work have been implemented in BALL as well. Already the alpha release of BALL has been successfully emplyed in about a dozen laboratories worldwide. The Max Planck Society honored the development of BALL with the Heinz Billing Award 2000 for the Advancement of Scientific Computation.

### Structure of the thesis

Following this introduction, Part II describes the new techniques for protein docking. First, we will give a short introduction to the biochemical concepts used in this work in Chapter 1. Chapter 2 gives an overview of protein docking and previous work in this field. We then present two algorithms for semi-flexible docking in Chapter 3 and an algorithm for NMR-based protein docking in Chapter 4. This part closes with a discussion of the presented docking algorithms in Chapter 5.

The design and the implementation of BALL is then described in Part III. Chapter 6 describes the architecture, design, and implementation of BALL. The techniques we used to manage the project and to ensure software quality are presented in Chapter 7. An example application, which illustrates the rapid software prototyping capabilities of BALL and its advantages over existing software, is given in Chapter 8. Chapter 9 discusses the current status of BALL and points out some developments planned for the future.

This work then concludes with a discussion in Part IV.

# Part II

# Protein Docking

Chapter 1

# Biochemistry – the Basics

This section gives an introduction to the biochemical terms used in this work. Since a serious introduction to this topic is beyond the scope of this work, we will just introduce the most important terms and refer to biochemistry textbooks for a more complete coverage of the subject (*e.g.* [74, 114]).

## 1.1 Atoms and Molecules

One of the oldest scientific models was proposed about 2500 years ago by *Leucippus of Miletus* and his disciple *Democritus of Abdera*. They deduced from a thought experiment that all matter should be composed of very small, undestructible entities. These fundamental particles were called *atoms* (from the greek $\alpha\tau o\mu o\sigma$, undivisible). Although the original model has been heavily modified and expanded in the course of history, it is still the most important and most useful concept in chemistry. According to this model, all matter – living or dead – is composed of atoms. Another, albeit younger, model is the *chemical bond*. A bond may be seen as a connection between atoms and the making and breaking of these bonds is what chemistry is all about. *Molecules* are groups of atoms with an exactly defined composition and topology. There are some smaller groups of atoms that are frequently found in molecules. These groups are usually called functional groups. Fig. 1.1 shows the structure of a well known molecule: *ethanol*. It contains a *hydroxyl group* (OH) that is characteristic for alcohols and a *methyl group* ($CH_3$).

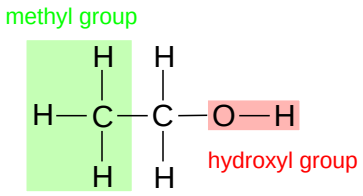



Figure 1.1: *Functional groups in ethanol*

Molecules can be grouped into families or classes of molecules according to their structure and the occurring functional groups. There are several classes of molecules that are characteristic for living beings – the so-called *biomolecules*. Among the most important classes of biomolecules are *proteins* and *nucleic acids*.

Proteins are the most important structural biomolecules – about half the dry mass of an animal cell consists of proteins. Nucleic acids are important as the carrier of hereditary information. Proteins as well as nucleic acids are *biopolymers*, *i.e.* they possess a linear, chain-like

structure. These chains are built from similar building blocks, the *monomers*. For proteins, these monomers are the *amino acids*.

## 1.2 Amino Acids

Amino acids (or more precisely $\alpha$-amino carboxylic acids) are small biomolecules. There are 20 amino acids that are commonly found. They differ only in their *side chain* (noted as R in Fig. 1.2). The common substructure they share contains two functional groups: the amino group ($NH_2$) and the carboxylic acid function ($COOH$). Both of these groups are bound to a central carbon atom, the $C_\alpha$ atom. In aquaeous medium, both groups tend to react with water, yielding a so called *zwitterion*. A zwitterion is an ion (a charged particle) that bears both a positive and a negative charge. The positive charge is caused by the amino group which accepts a proton ($H^+$) from the surrounding water, while the carboxylic acid function looses a proton and thus bears a negative charge.

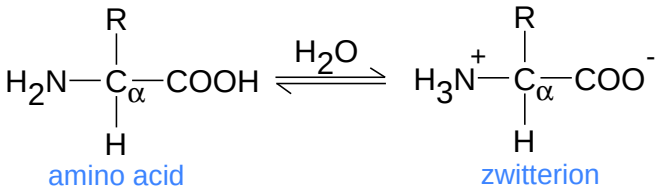



Figure 1.2: *Amino acids and their zwitterionic form*

## 1.3 Proteins

The amino group and the carboxyl group may also react with each other. In this reaction, a bond is formed between the nitrogen of the amino group and the carbon of the carboxyl group (Fig. 1.3) while a water molecule ($H_2O$) is liberated. This bond is called a *peptide bond*, the product is called a peptide. The product of the reaction possesses again an amino group and a carboxylic group, so further amino acids may be attached to both ends of the peptide. A peptide formed by two amino acids is termed *dipeptide*, one formed by three amino acids is a *tripeptide* and so forth.

While we use the term peptide for shorter chains, long chains of amino acids are called *proteins*. In this context, the amino acids are often referred to as *residues*. The end of the chain bearing the amino group is called the *N-terminus* while the carboxyl end is called the *C-terminus*. Another important term is the protein *backbone*. The backbone of a protein consists of a repeated sequence of three atoms of each residue: $C_\alpha$, $N$ (the amide N), and $C$ (the carbonyl C).

The *sequence* of a protein is defined as the sequence of its amino acids from the N-terminus to the C-terminus and is also called the protein's *primary structure*. Amino acid sequences are usually denoted as strings. There exist two different schemes to encode the amino acids: the

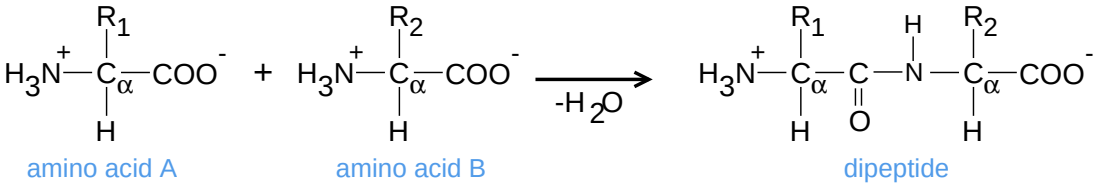



Figure 1.3: *The formation of a peptide bond*

*three-letter code* which denotes each amino acid with three letters and the shorter but less comprehensible *one-letter code*. In Fig. 1.5 an example for a protein sequence is shown.

Proteins also form *secondary structures*. This term refers to certain spatially repeating structures found in proteins. There are two types of secondary structures: the $\alpha$*-helix* and the $\beta$*-pleated sheet*. $\beta$-sheets are formed by parallel (or antiparallel) polypeptide chains (Fig. 1.4) that are bound to each other by hydrogen bonds between the backbone atoms.

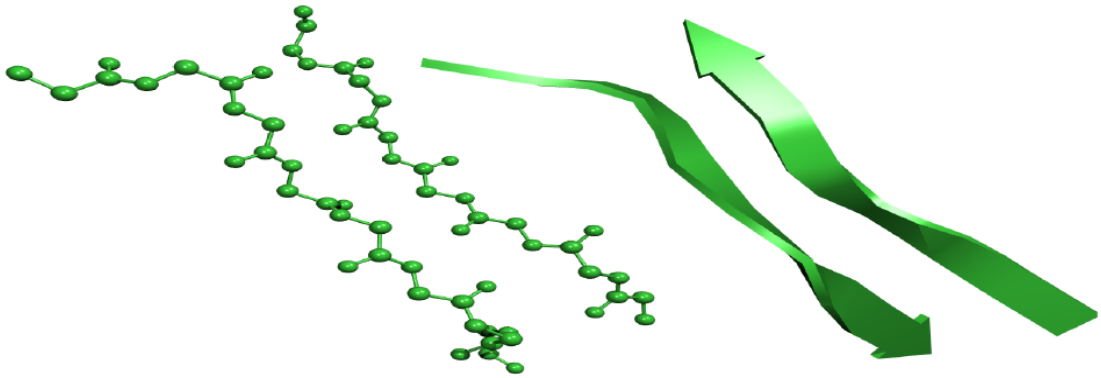


Figure 1.4: *A $\beta$-sheet is formed by two parallel polypeptide chains. They are often drawn as bands with arrow heads. For clarity, only the backbone of the two chains is shown.*

An $\alpha$-helix is a tight and regular helix of a polypeptide chain (Fig. 1.6). As in the $\beta$-sheet, this structure is stabilized by hydrogen bonds between backbone atoms. For a helix, these hydrogen bonds are not between different chains but between residues of the same chain. The helices are usually right-handed, although a left-handed variant exists as well.

Different sections of a protein chain may assume different secondary structures. The overall three-dimensional structure (fold) of a single protein chain (including all its secondary structure elements) is termed *tertiary structure*.

Proteins often aggregate to larger structures. With multiple polypeptide chains, *quarternary structure* is their interconnections and organization. The hierarchy of structures encountered in a large protein is shown in Fig. 1.5 on page 12.

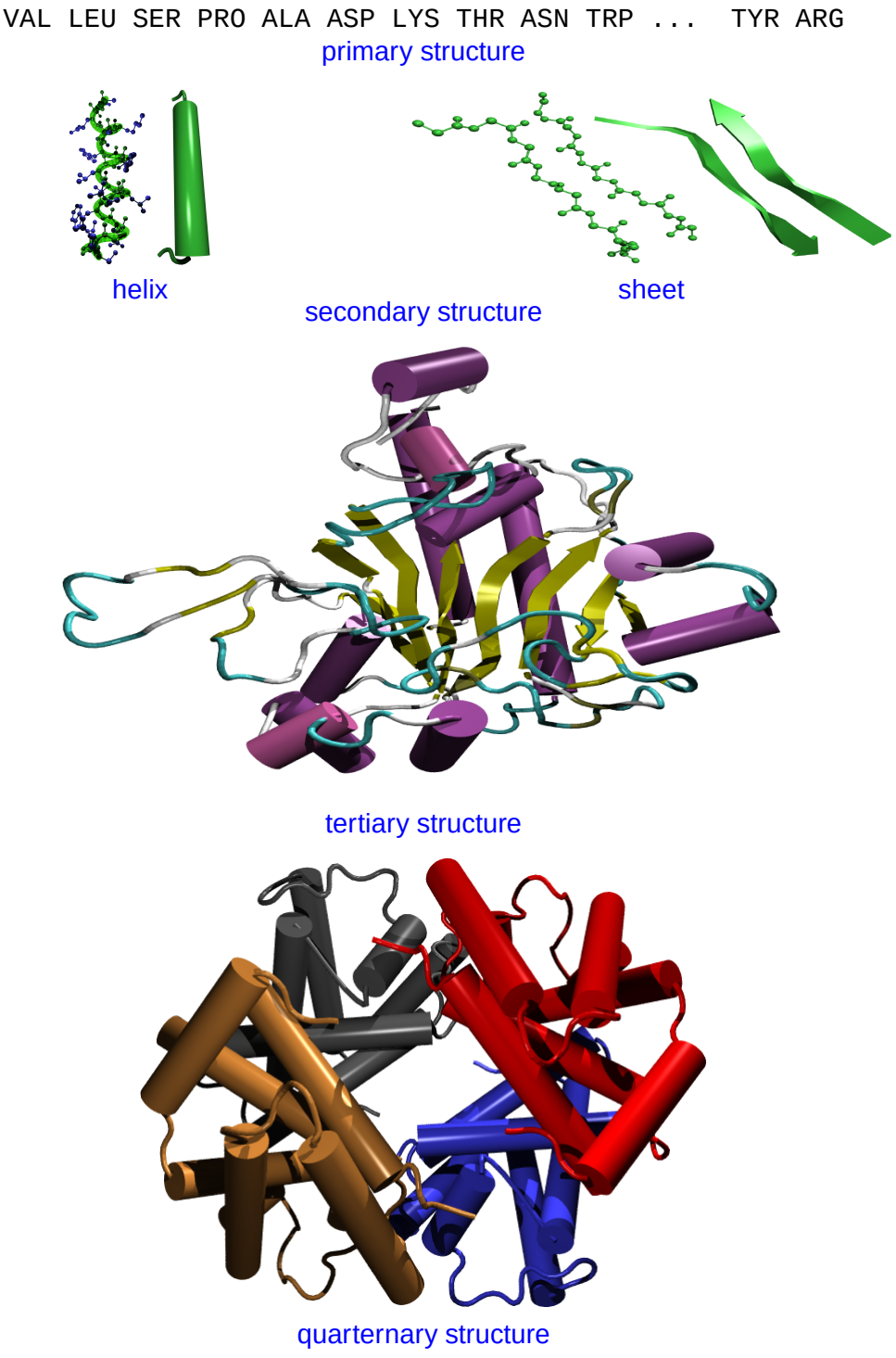



Figure 1.5: *The hierarchy of protein structures. The* primary structure *is defined as the amino acid sequence of the protein. Segments of a polypeptide chain may form* secondary structure elements*: helices or sheets. The total three-dimensional structure of a protein is termed* tertiary structure*. Finally, several proteins may aggregate to form* quarternary structures.

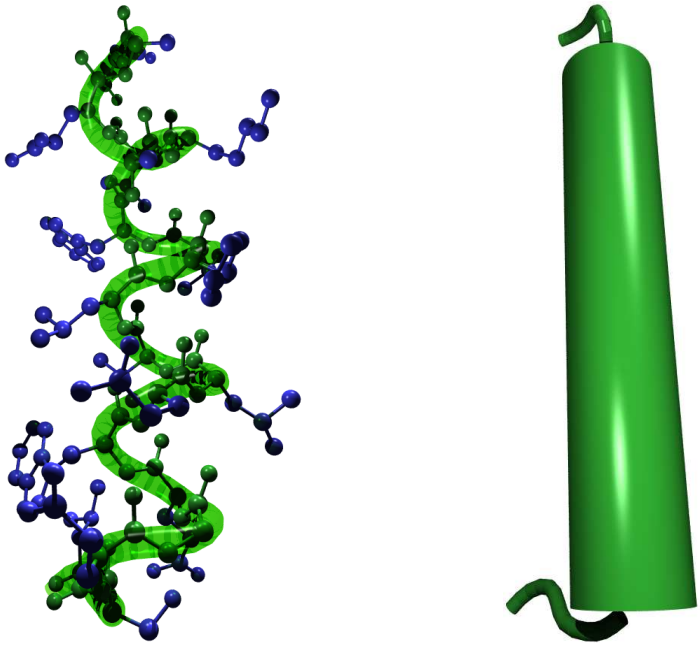


Figure 1.6: *In an $\alpha$-helix, the backbone of a polypeptide chain (shown in green, surrounded by a transparent tube) forms a right handed helix. In schematic illustrations of protein structure, helices are often represented by cylinders or tubes.*

## 1.4 Nucleic Acids

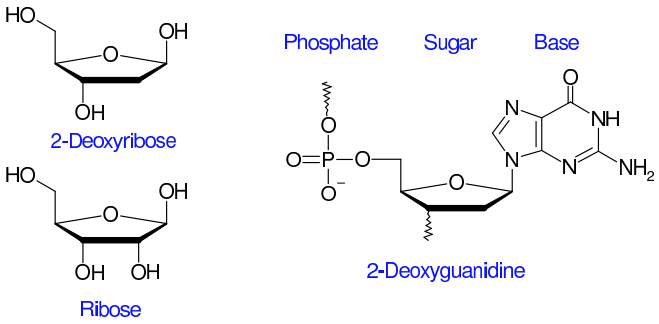



Figure 1.7: 2-Deoxyguanidine *(G) is the nucleotide composed of the base guanine, the sugar 2-deoxyribose, and a phosphate group.*

Nucleic acids are the carriers of hereditary information. Living organisms contain two kinds of nucleic acids: *ribonucleic acid* (RNA) and *deoxyribonucleic acid* (DNA). The monomeric building-blocks of DNA and RNA are the *nucleotides*.

A nucleotide consists of three subunits: a *sugar*, a *phosphate*, and a *base*. Fig. 1.7 shows this structure as well as the two occurring sugars: *ribose* which occurs in RNA and *deoxyribose* occurring in DNA.

There are five bases that can be found in nucleic acids (Fig. 1.8): *adenine* (A), *guanine* (G), *cytosine* (C), *thymine* (T), and *uracil* (U). Adenine, guanine, cytosine, and thymine represent the four letters of the DNA alphabet, while uracil occurs only in RNA.

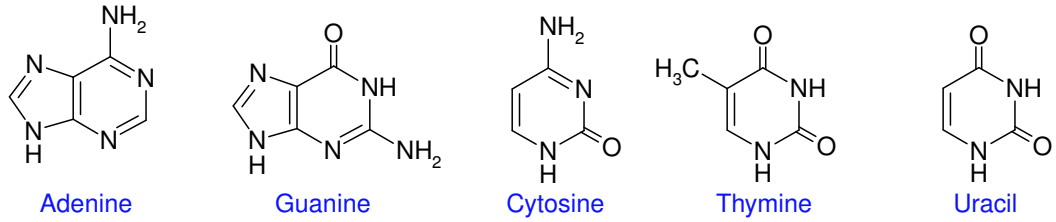



Figure 1.8: *The five bases occurring in DNA and RNA*

**DNA** molecules usually have a helical structure (Fig. 1.9). Unlike the $\alpha$-helices of proteins, the structure of DNA is a *double helix* formed by two DNA strands. This famous structure was discovered by Watson and Crick in 1953. The two strands of a chain are held together by hydrogen bonds. These bonds are formed between pairs of bases of opposite strands. Each base possesses a complementary base with which it may form a *base pair* (Fig. 1.9). Thus, each A is paired by a T in the opposite strand while C is complemented by G.

In **RNA**, thymine is replaced by uracil which is complementary to adenine as well. Thus, base pairing occurs in RNA as well. The sugar *2-deoxyribose* of the DNA backbone is replaced by another sugar: *ribose*. DNA and RNA differ also in their structure: RNA is usually single stranded. Instead of forming base pairs between two different strands, RNA tends to form intramolecular base pairs. This leads to very complex non-helical structures.

## 1.5 Interatomic Forces

The behaviour of molecules is ruled by a set of fundamental physical laws that describe the interaction of atoms. These interactions can be coarsely categorized as *bonded interactions* and *nonbonded interactions*. Bonded interactions are *intramolecular interactions*, *i.e.* they occur only between atoms of the same molecule, as they are mediated by bonds. In contrast, nonbonded interactions are not mediated by bonds and can thus occur between atoms of different molecules as well.

### 1.5.1 Nonbonded interactions

The existence of intermolecular, hence nonbonded, interactions can be deduced by observing a cup of tea. The fact that the tea stays in the cup is due to attractive intermolecular forces. These attractive forces make the water molecules be attracted to each other and thus form a liquid instead of a gas. Only those water molecules with a high energy evaporate as steam,

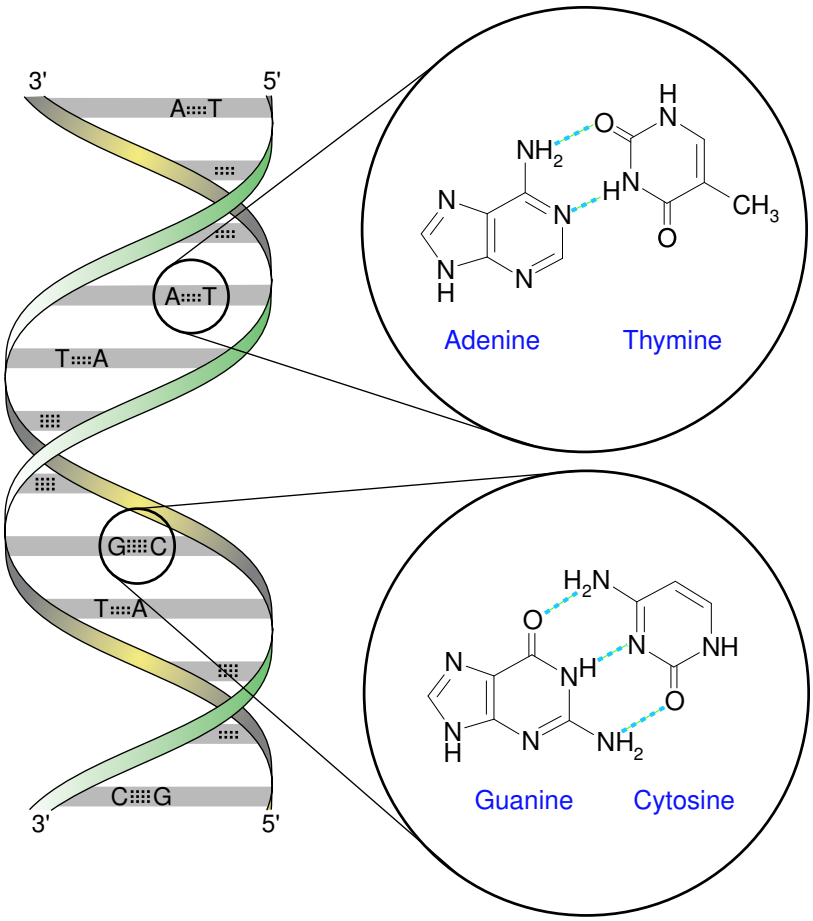



Figure 1.9: *Double-stranded DNA assumes the structure of a double helix. The structure is stabilized by pairs of complementary bases. These pairs form two (A-T) or three (C-G) hydrogen bonds.*

as they contain a sufficient amount of energy to overcome the attractive interactions. Not all intermolecular forces are attractive. The fact that water has a well defined volume and is not easily compressed is due to repulsive interactions that prevent the molecules from coming too close to each other.

### Electrostatic interactions

One of the most important nonbonded interactions are the electrostatic interactions, which occur between electrostatic charges. Atoms consist of charged elementary particles (protons and electrons). Protons bear a positive charge, while the electrons bear a negative charge of equal magnitude. If atoms or molecules contain a different number of electrons and protons, they are called *ions* and they bear a positive or negative net charge. But also molecules with equal numbers of protons and electrons may still have a charge distribution that leads to so-called *partial charges*, *i.e.* regions of positive or negative charge excess. The interaction of

these electrostatic charges can be described by *Coulomb's law*:

$$E_{\mathrm{ES}} = \frac{1}{4\pi\varepsilon_0}\frac{q_1 q_2}{r} \tag{1.1}$$

where $E_{\mathrm{ES}}$ is the energy resulting from the electrostatic interaction of two charges $q_1$ and $q_2$ with distance $r$ between the charges. $\varepsilon_0$ is the permittivity of the vacuum, a constant. Since the *force* between the charges can be derived from the energy as the negative gradient, we can also calculate the forces between the two charges:

$$\vec{F}_{\mathrm{ES}} = -\nabla E_{\mathrm{ES}} \tag{1.2}$$

$$= -\frac{1}{4\pi\varepsilon_0}\frac{q_1 q_2}{r^3}\vec{r} \tag{1.3}$$

Depending on the sign of the charges $q_1$ and $q_2$, electrostatic forces can be either attractive (opposite signs of the charges) or repulsive (same sign).

#### Van der Waals interactions

The term *van der Waals interactions* describes nonbonded interactions that consist of an attractive and a repulsive part. The attractive part stems form induced dipole–induced dipole interactions, *i.e.* fluctuations of the charge distribution in one atom or molecule induce charge fluctuations in a neighboring atom. These charge fluctuations lead to an attractive electrostatic interaction. The repulsive part results form the *pauli exclusion principle*, a quantum mechanical effect that results in unfavorable energies for interpenetrating electron coulds of two approaching atoms. The interplay of attractive and repulsive interactions leads to intermolecular potential functions like that shown in Fig. 1.10. For large distances, the energy approaches zero. At intermediate distances, the energy is negative, which leads to attractive forces. If the distance between the atoms is further reduced, the repulsive forces grow rapidly and give highly positive energies. There are different models to describe that kind of potential. The most commonly used expression uses two parameters $A$ and $B$ that depend of the type of the atoms involved to describe the van der Waals energy as a function of the atom distance $r$:

$$E_{\mathrm{vdW}} = \frac{A}{r^{12}} - \frac{B}{r^6} \tag{1.4}$$

### 1.5.2 Molecular Mechanics

Molecular Mechanics is an approach based on simple physical models of molecular and atomic interactions. The parameters of these models are usually obtained by fitting to experimental data. The resulting set of equations describing the interactions and the corresponding parameters are called a *force field*. Besides the nonbonded interactions described in the previous section, the force fields also include bonded interactions, which can be modeled in a number of different ways.

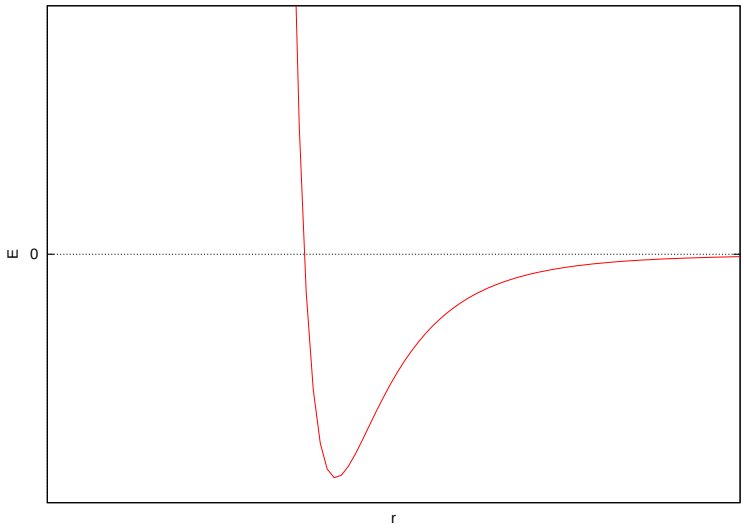



Figure 1.10: *The distance dependence of the van der Waals energy.*

A typical force field like the AMBER force field defines the total energy of a set of molecules as the sum of five different types of interactions:

$$
\begin{aligned}
E_{total} &= E_{\mathrm{vdW}} + E_{\mathrm{ES}} + E_{\mathrm{bend}} + E_{\mathrm{stretch}} + E_{\mathrm{torsion}} \\
&= \sum_{i<j} \left( \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}} \right) \\
&+ \frac{1}{4\pi\varepsilon_0} \sum_{i<j} \frac{q_i q_j}{r_{ij}} \\
&+ \sum_{(i,j)\in \mathrm{bonds}} k_{ij}^{\mathrm{stretch}} (r_{ij} - r_{ij}^0)^2 \\
&+ \sum_{a\in \mathrm{angles}} k_a^{\mathrm{bend}} (\theta_a - \theta_a^0)^2 \\
&+ \sum_{b\in \mathrm{torsions}} k_b^{\mathrm{torsion}} (1 + \cos(n_b \phi_b - \phi_b^0))
\end{aligned}
$$

The *van der Waals energy* $E_{\mathrm{vdW}}$ and the *electrostatic energy* $E_{\mathrm{ES}}$ are the nonbonded interactions. The *stretch energy* $E_{\mathrm{stretch}}$ describes the change in the energy as the bond distance $r_{ij}$ varies. This energy is described by a harmonic potential (*i.e.* a quadratic function of the deviation from the optimal bond distance, see Fig. 1.11). Similarly, the *bend energy* $E_{\mathrm{bend}}$ describes the variation of the energy with the bond angle $\theta$ formed by two neighboring bonds. This contribution is also described by a harmonic potential. The contribution $E_{\mathrm{torsion}}$ describes the *torsion energy*. Torsions describe the variation of the total energy on rotation about a bond. The torsion energy depends on the *torsion angle* $\phi$. It is defined as the angle that is enclosed by the two planes defined by atoms $(i, j, k)$ and $(j, k, l)$ (see Fig. 1.11). The variation of the energy with this angle can be described as a cosine function.

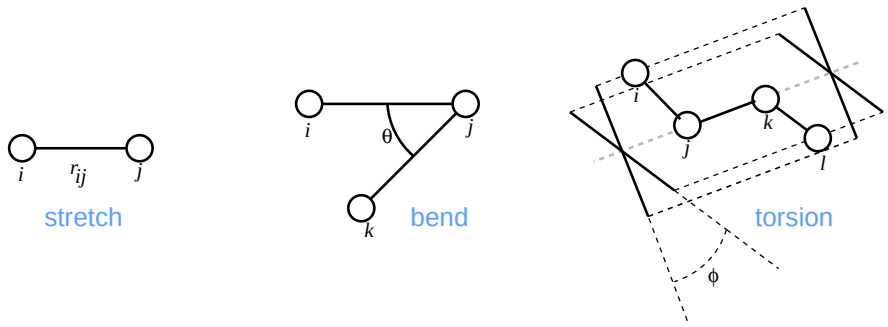



Figure 1.11: *The bonded interactions in a typical Molecular Mechanics force field.*

This form of the force field describes the energy as a function of the coordinates. For many applications it is also necessary to know the forces caused by these interactions. The force $\vec{F}_i$ acting upon atom $i$ may be derived as the negative gradient of the energy:

$$\vec{F}_i = -\nabla E = -\nabla E_{\mathrm{vdW}} - \nabla E_{\mathrm{ES}} - \nabla E_{\mathrm{stretch}} - \nabla E_{\mathrm{bend}} - \nabla E_{\mathrm{torsion}} \tag{1.5}$$

The force field permits the calculation of the static and dynamic properties of molecules or sets of molecules. There are numerous applications for force fields. Among the most important ones are energy minimization and Molecular Dynamics simulations. Energy minimization means to search for the molecular geometry with the lowest energy, a very difficult multi-dimensional optimization problem that is usually tackled using gradient-based optimization techniques. Molecular Dynamics is based on the fact that the force field gives a full description of the forces acting upon the atoms. If these forces can be calculated for arbitrary geometries, one can apply Newton's equations of motion and thus simulate the dynamic behaviour of molecules.

Chapter 2

# Introduction

Since the protein docking problem was already introduced in Part I, we will only briefly repeat the problem definition here: We are given the three-dimensional structure of two proteins $A$ and $B$, that are known to form a complex $AB$. The protein docking problem is then to predict the structure of the complex $AB$.

The protein docking problem has numerous interesting applications. Besides the standard problem (how does the complex structure look like?), it may also speed up the time-consuming process of structure elucidation (see Chapter 4). The results of docking predictions are also useful for the analysis and the understanding of the binding modes of proteins and numerous new applications will arise with the coming advent of Proteomics. The following sections will briefly discuss existing approaches for protein docking, their limitations, and some possibilities for improvement.

## 2.1 Rigid Body Docking

Rigid protein protein docking is based on a rather old analogy by German chemist Emil Fischer, the so-called *lock-and-key principle* he proposed in 1894:

> "Invertin und Emulsin haben bekanntlich manche Aehnlichkeit mit den Proteïnstoffen und besitzen wie jene unzweifelhaft ein asymmetrisch gebautes Molekül. Ihre beschränkte Wirkung auf die Glucoside liesse sich also auch durch die Annahme erklären, dass nur bei ähnlichem geometrischem Bau diejenige Annäherung der Moleküle stattfinden kann, welche zur Auslösung des chemischen Vorganges erforderlich ist. **Um ein Bild zu gebrauchen, will ich sagen, dass Enzym und Glucosid wie Schloss und Schlüssel zu einander passen müssen**, um eine chemische Wirkung aufeinander ausüben zu können. [1]" *[34], p. 2992*

This model implies that enzyme and substrate (or in our case: the two proteins) are rigid bodies that possess regions of geometric complementarity. A contemporary view of this model is shown in Fig. 2.1. In the mean time, the lock-and-key model has been superceded by a number of various other models, *e.g.* the *induced fit* hypothesis by David Koshland [64]. However, it is still of fundamental importance and for many protein-protein interactions it remains a valid assumption, as has been shown in a recent study by Betts and Sternberg [11].

The first docking algorithms were based on the assumptions of rigidity and complementarity (thus the name *rigid-body docking*, RBD). They tried to identify geometrically complementary sections of the proteins $A$ and $B$. These regions should define the binding site of the com-

[1] English translation: "Invertin and emulsin are known to share some similarities with protein compounds and, like those, possess an asymmetric molecular structure. Their limited effect on glycosides could thus be explained by the hypothesis that the approachment that is necessary to initiate the chemical process is only possible if the molecules are geometrically similar to each other. **To use an image, I would say that enzyme and glycoside have to fit into each other like a lock and a key**, in order to exert a chemical effect on each other."

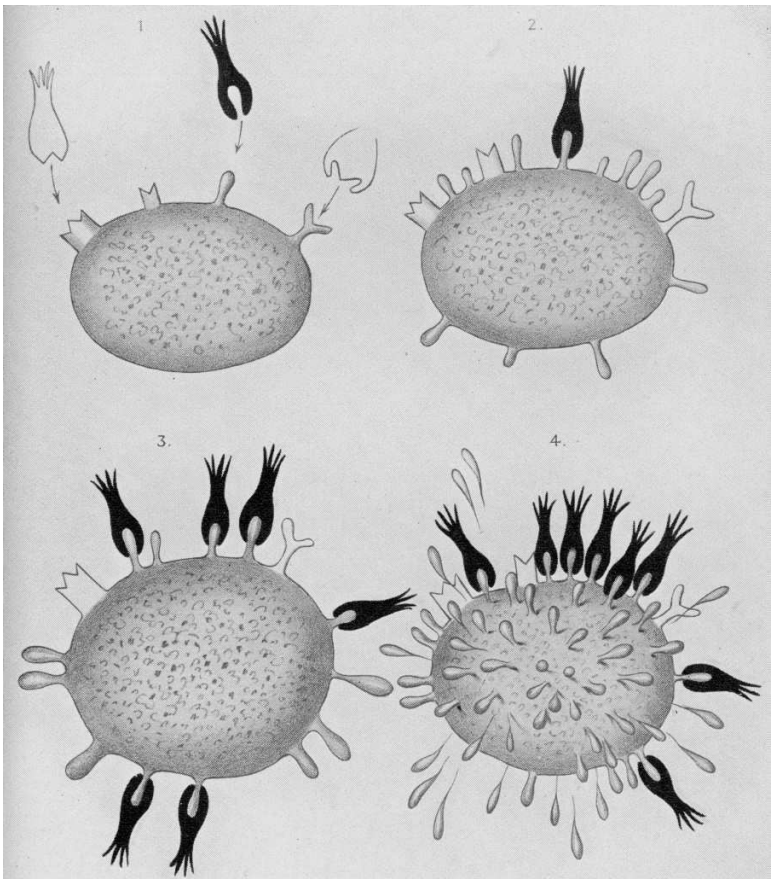



Figure 2.1: *Paul Ehrlich applied Fischer's lock-and-key principle to immunoreactions. He illustrated his notion of the concept in this drawing from 1900 [29].*

plex. For example, the correlation-based techniques introduced by Katchalski-Katzir *et al.* [60] estimate the contact area of $A$ and $B$ and assign those structures with the largest contact area the best scores. These purely geometric algorithms (*e.g.* [20, 125, 33, 32]) were successful for a large number of examples.

It was soon recognized, that geometric complementarity alone is not sufficient to identify the binding site. The next generation of algorithms thus employed simple energy functions to estimate the binding free energy of the resulting complexes. These energy functions usually consider only a part of the physics involved in the binding process, *e.g.* only electrostatic interactions [40, 119] or hydrogen bonds [80].

Today, most RBD algorithms share a similar overall structure (Fig. 2.2, page 22). In a first step, a large number of potential complex structures is created from the two protein structures $A$ and $B$. This generation step is usually based on geometric complementarity. For example, in the algorithm by Lenhof [70], this structure generation is based on the matching of triangles of surface points of $A$ and triangles formed by atom centers of $B$. A pair of triangles from $A$ and $B$ matches, if the side lengths of the triangles coincide within a certain tolerance. Each of

these triangle pairs defines a rigid transformation that brings $B$ into close contact with $A$. This structure generation step can be implemented very efficiently using geometric hashing techniques and can produce huge amounts of tentative complex structures in a short time (usually several thousands). In the vast majority of cases, this set of candidates contains a number of good approximations of the true complex structure (where "good" means that the RMSD[2] of all atoms is below $3$ Å).

The next (and clearly the most difficult) task is the identification of these good approximations among the candidates. We will refer to them as *true positives*, while the remaining structures (those that do not resemble the true complex structure) will be called *false positives*. Numerous methods have been proposed to distinguish true positives from false positives. In principle, it were sufficient to calculate the binding free energy of each of the tentative complex structures, as the true positives should possess the lowest binding free energy. However, this is a non-trivial task, since only very rough approximations are known to estimate the free energy on binding.

A large number of other scoring functions have been tested for the protein docking problem, but the correct prediction of binding free energies is still one of the most difficult problems in this field. We cannot enumerate all these methods, so we refer to the comprehensive review by Sternberg *et al.* [113].

In all algorithms, one or more of these scoring functions are applied to the initial set of candidates. The output of each docking algorithm is thus a ranked list of structures, where the good approximations should be at the top of the list (bottom of Fig. 2.2). If the generation step produced good approximations of the true complex structures and the scoring function were ideal, one should expect that the candidate with the best score were a good approximation of the true complex structure. One measure of the algorithm's quality is thus the rank assigned to the first true positive in the list.

## 2.2 Docking and Protein Flexibility

RBD algorithms perform rather well as long as the structure of the two proteins does not change significantly upon complex formation. Unfortunately, there are numerous examples where this assumption does not hold. They are explained with Koshland's *induced fit* hypothesis, which basically states that the two partners change their shape upon binding and form a complex structure where they possess geometric complementarity. In contrast to the lock-and-key principle, the induced fit hypothesis does not require the unbound structures of $A$ and $B$ to display geometric complementarity. Clearly, this leads to a more difficult docking problem. Those examples that show structural changes on binding fall essentially into two categories:

- domain movements
- side chain movements

[2] root mean square deviation

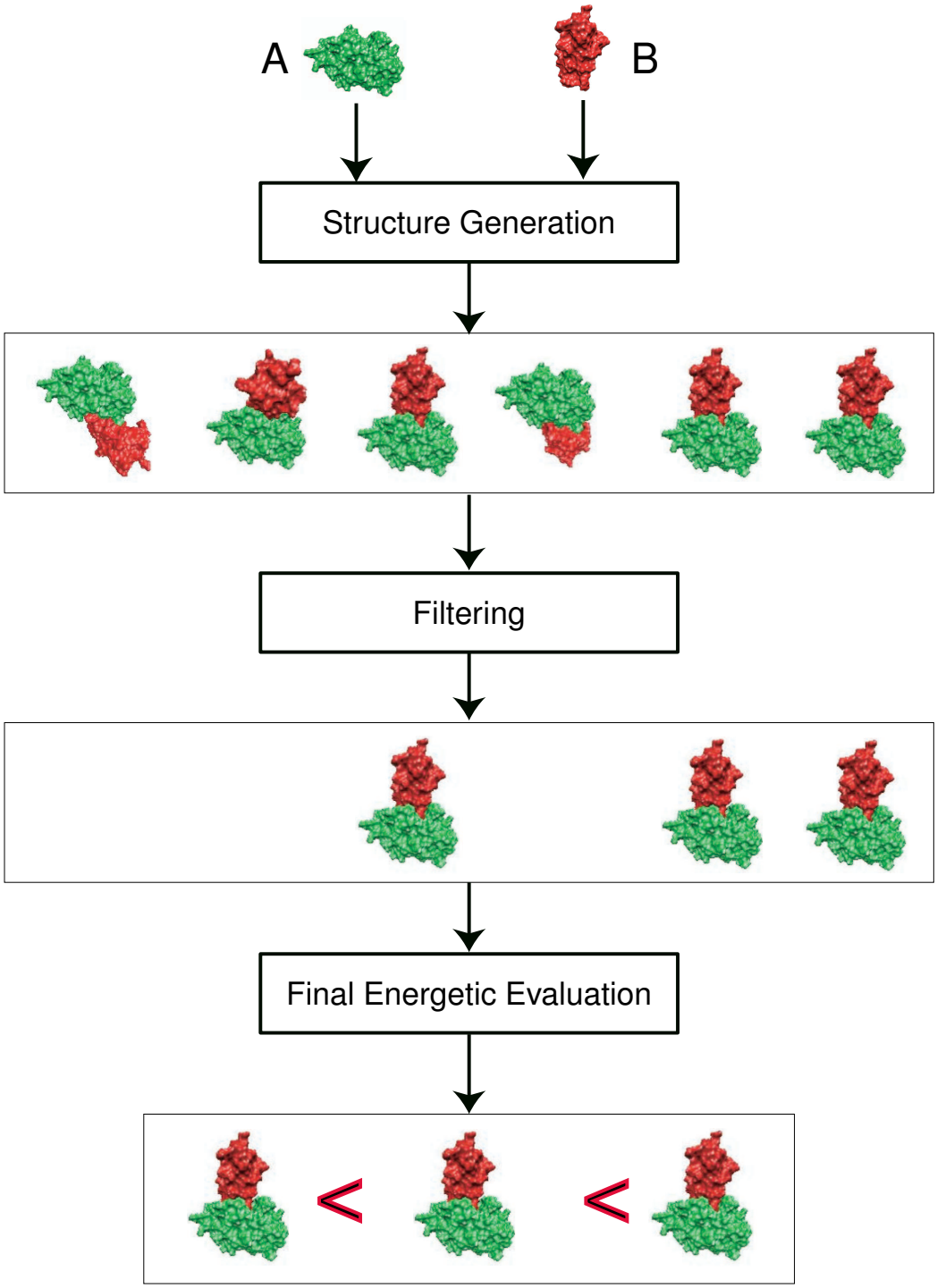



Figure 2.2: *The overall structure of a rigid-body docking algorithm.*

### Domain movements

In the first case, large rigid sections of the protein (domains) move around “hinges”, flexible joints tethering the domains and restricting their movement. The hinges are formed by small regions of high backbone flexibility, while the backbone remains essentially rigid inside the domains. In the case of domain movements, one of the major difficulties is to distinguish between rigid and flexible parts of the protein. Also the total number of hinges that can be considered is usually very limited due to the number of degrees of freedom associated with each hinge. One way to handle domain movements was proposed by Sandak *et al.* [105, 106]. Their algorithms performs a search of the full three-dimensional rotational space of a hinge between two domains. To identify the best structure, a number of potential hinge locations is tested and a geometric docking is carried out. Finally, all structures resulting from all possible hinge positions are ranked with respect to geometric complementarity.

### Side chain flexibility

In the second and more frequent case, the protein backbone remains essentially rigid, but side chains undergo conformational changes on binding. An example for this category is the complex of trypsin and BPTI, as we have already discussed in Part I (page 2). Norel *et al.* [89] studied the bound and unbound structures of 26 protein complexes and could show that the protein backbones remain rigid upon binding and only the side chains show significant conformational changes for the majority of all examples.

RBD approaches have little hope for success if domain or side chain movements occur. This is mainly due to the fact that the two unbound structures (which represent the input of the docking algorithm) do not possess the geometric complementarity the bound structures have. Even though the initial candidate set often contains a large number of true positives, they are usually not ranked correctly. The failure of the scoring functions stems from the fact that side chains of $A$ and $B$ overlap for many true positives. The resulting structures cannot occur in nature, they are physically meaningless. Hence, the scoring functions are not able to give a reasonable estimate of the true binding energy.

Several approaches have been proposed to overcome these difficulties. Common to these approaches is an intermediate stage, where the side chain conformations of the initial candidates are demangled to produce physically meaningful structures. In Chapter 3 we present two new approaches to protein docking with flexible side chains. The first technique uses a greedy heuristic to place the side chains, while the second approach is a branch-&-cut algorithm that yields optimal solutions to the side chain placement problem.

## 2.3 Combining NMR Data and Docking Algorithms

Since it is not yet possible to predict the free energies on binding with sufficient accuracy, other ways have to be found to increase the reliability of the docking results. This can be done by counterchecking the results against experimental data. This experimental data should be easily accessible and should allow a ranking of the docking results or at least some way of validation.

Nuclear Magnetic Resonance (NMR) spectroscopy is one of the two important methods of protein structure elucidation. It provides data that contains a large amount of structural information. This information has been used in ligand docking for quite some time to predict the structure of protein-ligand complexes [127, 44, 98]. With the fundamentals of NMR spectroscopy being discussed in depth in Chapter 4, we will only briefly explain these methods here. There are different kinds of structural information available from NMR data. The easiest to obtain are simple *NMR spectra*. These spectra measure the so-called *chemical shift*, an intrinsic property of each atom of the protein. This shift depends on the structural environment (electronic surrounding, geometry, neighboring atoms, *etc.*) of the atom, hence the value of the chemical shift contains structural information. However, due to the large number of atoms showing in such a spectrum (typically several hundreds), it is a non-trivial task to decide which of the peaks in the spectrum belongs to which atom. This task is also called *shift assignment* and is the most time-consuming task in NMR-based structure elucidation, which can often take several months. All of the above mentioned protein-ligand docking methods are based on fully assigned spectra. From these spectra, geometric constraints (NOE[3] constraints) are derived. The methods are then based upon the integration of these distance constraints into existing docking techniques and the resulting structures have to satisfy as many of these constraints as possible. A similar methodology was recently proposed by Morelli *et al.* [82] for protein-protein docking. They used NOE constraints in a soft docking algorithm to predict the structure of the complex of cytochrome $c_{553}$ and ferredoxin.

In contrast to these constraint-based docking methods, our algorithm does not require a shift assignment. This extends the applicability of the method to use structures where no NMR shift assignment is known (*e.g.* structures determined by X-ray methods). Furthermore, we only need an unassigned $^1$H-NMR spectrum of the complex, which is the simplest kind of spectrum and thus easy and cheap to obtain. Instead of processing the spectra (which has yet to be done manually) and extracting geometric information from the experimental data, we chose the opposite approach. We process the candidates generated from a RBD algorithm and predict their spectra. These spectra are then compared to the experimental spectrum and ranked according to their similarity. Hence, no prior processing and evaluation of the experimental data is required. Chapter 4 gives a short introduction to the basics of NMR spectroscopy and then explains our new approach in more detail.

[3]NOE – *nuclear Overhauser effect*. This effect permits the estimation of the distance between neighboring atoms.

Chapter 3

# Semi-Flexible Docking

## 3.1 Introduction

The previous chapter introduced the basic techniques for rigid body protein docking and their limitations. Since completely flexible protein docking is computationally very expensive, the use of flexible side chains is a reasonable compromise. We call the docking with rigid backbones and flexible side chains *semi-flexible docking*. Algorithms for semi-flexible docking are similar to the algorithms for rigid-body docking. They introduce an additional step, the *side chain demangling*. In this step, the side chain orientations in the binding site of each tentative complex structure are optimized and occurring overlaps are thus removed. This demangling step ensures physically meaningful structures that are then subjected to a final energetic evaluation and ranking. The next sections will first discuss existing techniques for side chain placement and then we will describe our new approaches and some results obtained with these techniques. Parts of this chapter have been previously published in [2, 3, 4].

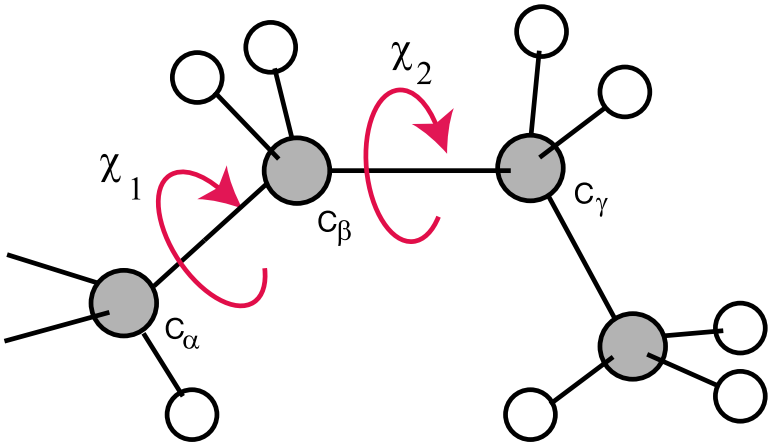



Figure 3.1: *This figure illustrates torsion angles in an amino acid side chain. The torsion angles determine the rotation about single bonds in the side chain. The first two torsion angles in a side chain are usually denoted* $\chi_1$ *and* $\chi_2$.

First, we have to define what side chain flexibility means. In principle, the $N$ atoms of a side chain have $3N$ translational degrees of freedom. In protein structures, we usually observe that bond lengths and bond angles deviate only slightly from the ideal values observed in structures of minimal energy. The reason is that the energy needed to stretch a bond or deform a bond angle is much larger than the energy required to perform rotations around the

bond. Therefore, the side chain's main source of flexibility are torsions around single bonds (see Fig. 3.1). A side chain has a small number of these torsional degrees of freedom (between zero and five torsion angles depending on the amino acid). Restricting side chain flexibility to torsional flexibility drastically cuts down the complexity of the side chain placement problem.

In 1987, Ponder and Richards [99] observed that the side chain conformations occurring in proteins can be adequately described by a rather small set of so-called *rotamers* for each amino acid. Fig. 3.2 shows a so called *Ramachandran plot*. It was obtained by examining a set of known protein structures. For each lysine side chain occurring in the structure, the first two side chain torsion angles $\chi_1$ and $\chi_2$ were determined. The plot shows the observed frequency of these angles. Obviously, not all possible angle combinations are assumed with equal probability. In fact, there are only three small and well-defined regions whose angle combinations occur frequently in proteins. These regions are small enough to assume that the conformations in a region are energetically equivalent. Therefore, Ponder and Richards argued that it is sufficient to describe the conformational space of a side chain through a set of conformations, the so-called rotamers. In our example, the rotamers would correspond to the conformation obtained by picking the three maxima in the Ramachandran plot. All rotamers of the amino acid side chains form a so-called *rotamer library*.

The use of the rotamer representation for the side chain conformations reduces the initial continuous optimization problem to a discrete combinatorial optimization problem: we have to identify the set of rotamers with the minimum energy, *i.e.* the *global minimum energy conformation* (GMEC).

In principle, each side chain can assume any of its rotamers, hence the total number of combinations is the product of the number of rotamers for each side chain. With the typical number of side chains in a binding site ranging from 40 to 60, the total number of possible rotamer combinations can become as high as $10^{60}$ combinations. Thus, efficient algorithms are required to identify the GMEC or suboptimal solutions sufficiently close to the GMEC.

Side chain placement has been discussed in depth not only for the *ab initio* prediction of protein structures and homology modeling. There is a wide range of techniques available for side chain placement. Due to the high dimensionality of the conformational space of all side chains, exhaustive search is only tractable for a very small number of side chains [16, 128]. Other approaches use Monte Carlo methods [1, 50], local homology modeling [65], or self-consistent field theory [62] to place the side chains.

Methods for side chain demangling were also successfully applied to protein docking. Totrov and Abagyan [117] used a Monte Carlo approach to predict the structure of a lysozyme-antibody complex. Weng *et al.* [128] optimized the side chains in the binding site of three protease-inhibitor complexes using exhaustive search. Jackson *et al.* [53] employed a self-consistent mean field approach and a Langevin dipole model for the side chain placement in several protein complexes.

However, no algorithm has been yet presented that employs an optimal side chain placement for the interface refinement in protein-protein docking except for the exhaustive search of Weng *et al.* [128], which is severely limited in the number of side chains and thus not tractable for larger protein interfaces.

The only algorithm that is able to solve the side chain placement problem to optimality for a related problem (protein-ligand docking) was proposed by Leach [67, 69]. Leach first employs

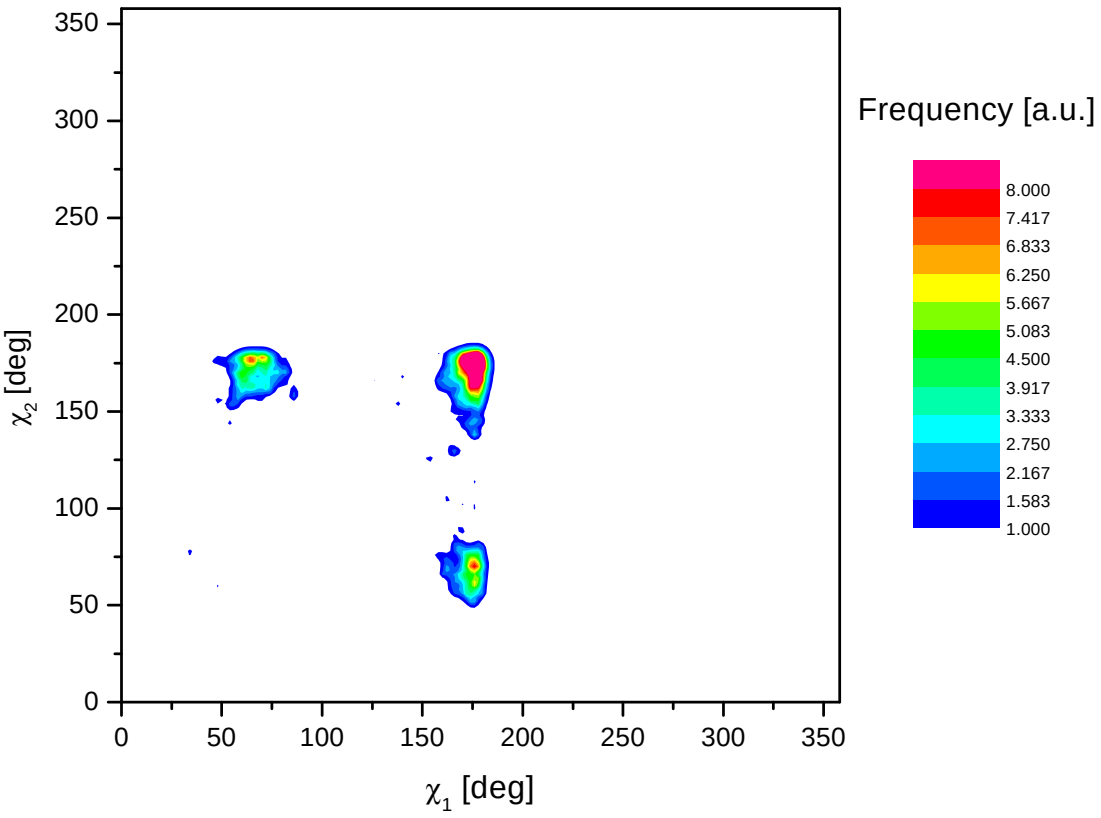



Figure 3.2: *Ramachandran plot for the $\chi_1$ and $\chi_2$ torsion angles of LYS. The color coding represents the frequency of occurrence in a protein test set in arbitrary units.*

the Dead End Elimination theorem to reduce the complexity of the placement problem and then the A* algorithm to determine an optimal placement of the side chains of the receptor with respect to a Molecular Mechanics force field.

We present two new techniques for protein interface refinement and their application to the docking of unbound protein structures. First, we present a fast heuristic that usually yields suboptimal solutions to the side chain placement problem. Nevertheless, the solution is close enough to the optimal solution to allow the correct prediction of the true complex structures for a test set of unbound protein structures. Second, we propose the first algorithm that allows the optimal, albeit slower, demangling of side chains of large protein interfaces. This method is based on the formulation of the side chain placement problem as an integer linear program and its solution using a branch-&-cut algorithm. The results of experiments with three protease-inhibitor complexes are presented in Section 3.3.

## 3.2 The Docking Algorithm

The input of the docking algorithm consists of two proteins $A$ and $B$ in their unbound conformation. These proteins are subjected to a rigid docking, yielding a set of candidate conformations. For the 60 best candidates out of this set, we perform a side chain demangling. Finally, the candidates are scored according to their binding free energy.

### 3.2.1 Rigid Docking

A rigid docking is performed for the proteins $A$ and $B$, using an improved version of the algorithm described in [70] and [71]. The algorithm uses geometric complementarity and simple chemical fitness functions to create a large set of potential complex structures. Out of this set, the 60 best candidates with respect to geometric complementarity are selected. An experiment with a test set of docking structures (35 complexes) always produced several good approximations of the true complex structure among the 60 best candidates.

### 3.2.2 Side Chain Demangling

The rigid docking algorithm, which generates a set of promising candidates, neglects the side chain flexibility. Therefore, many of the candidates have strong overlaps and incorrectly placed side chains. In order to obtain physically meaningful conformations, we have to demangle the side chains of the protein interface (the binding site).

#### Determination of the binding site

First, we have to identify all residues of $A$ and $B$ that belong to the binding site. We consider a residue to be part of the binding site if any of its atoms is within $6$ Å of any atom of the other protein. All residues that fulfill this condition are marked and kept in a list $BS$. Side chains without rotamers (CYS engaged in disulfide bonds, ALA, and GLY) are excluded from this list. For each residue in this list, we determine its set of possible rotamers from the rotamer library of Dunbrack *et al.* [28], assuming that the side chain can occur only in one of these conformations. We have now decomposed the protein residues into two disjoint sets: one set comprises all residues that have rotamers and belong to the binding site as defined by our cut off criterion, and the other set comprises all remaining residues (= the template).

#### Decomposition of the total energy

The GMEC is now defined as the combination of rotamers yielding the lowest total energy. This entails a combinatorial optimization problem where we must search a huge conformational space for the GMEC. In order to demangle the side chains of the binding site, we have to identify the GMEC or a good approximation thereof. By the following decomposition we can express the total energy of the system as a function of the selected rotamers:

$$E^{total} = E^{tpl} + \sum_i E^{tpl}_{i_r} + \sum_i \sum_{j<i} E^{pw}_{i_r,j_s} \tag{3.1}$$

where $E^{tpl}$ is the potential energy of the template (= system without residues of the binding site) and $E^{tpl}_{i_r}$ is the potential energy of side chain $i$ in rotameric state $r$ (short: rotamer $i_r$) interacting with the atoms of the template (including the internal energy of $i_r$). $E^{pw}_{i_r,j_s}$ is the pairwise potential energy between side chain $i$ in rotameric state $r$ and side chain $j$ in rotameric state $s$. This decomposition is exact, as the AMBER force field contains only pairwise nonbonded interactions. Since rotamers that heavily overlap with the rigid template cannot be part of the GMEC, we remove all rotamers whose interaction energy with the template is larger than $100\ \mathrm{kJ/mol}$.

For a given set of rotamers, the single energy components of eq 3.1 can be easily calculated via the Molecular Mechanics force field. The computational effort required for the evaluation scales at most quadratically with the number of atoms (there are at most $N^2$ nonbonded atom pairs for $N$ atoms). The difficulty arises in finding the global minimum as this means picking the optimal combination of rotamers for the side chains. Fortunately, the search space can be significantly reduced by the so-called *dead-end elimination theorem* (DEE)[25], which can be stated as follows: if for a pair of rotamers $(i_r, i_t)$,

$$E^{tpl}_{i_r} + \sum_{j \neq i} \min_s E^{pw}_{i_r,j_s} > E^{tpl}_{i_t} + \sum_{j \neq i} \max_s E^{pw}_{i_t,j_s} \tag{3.2}$$

holds, then the rotamer $i_r$ can be safely ignored in the search for the global minimum. Iteratively applying the DEE theorem reduces the number of combinations of rotamers by several orders of magnitude (see also the table in Section 3.3 on page 40).

#### Search of minimum energy conformation

In order to find an optimal combination of rotamers, we have developed two alternative approaches: a tree-based multi-greedy (MG) method and a *branch-&-cut algorithm* based on an ILP (Integer Linear Programming) formulation of our problem. We will first describe the greedy method and then explain the more technical branch-&-cut algorithm.

### 3.2.3 The Multi-Greedy Method

In this approach, an enumeration tree representing all possible rotamer combinations is built. The tree consists of $k = |BS|$ layers, each representing one side chain. Each path from the root to a leaf represents a possible combination of rotamers. The label of the node in layer $i$ determines which rotamer is selected for side chain $i$. Every node also possesses an energy label. For the construction, we start out with an artificial root node $v$, which is given an energy label $E(v) = E^{tpl}$ and a rotamer label $r(v) = null$. For each side chain $i$, we add a new layer to the tree. This layer is constructed by adding a node for each rotamer of $i$ to each leaf of the previous layer (see Figure 3.3). Each new node $x$ for a rotamer $i_s$ is labeled with the corresponding rotamer ($r(x) = i_s$) and its energy label is set to

$$E(x) = E^{tpl}_{i_s} + E(parent(x)) + \sum_{y \in pred(x)} E^{pw}_{i_s,r(y)},$$

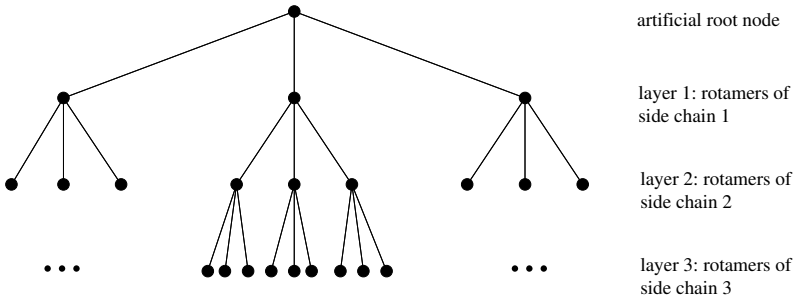



Figure 3.3: *The tree for three side chains, each having three rotamers.*

where $pred(x)$ denotes the set of nodes that lie on the path from $x$ to $v$ ($x$ excluding) in the tree. Thus, each node is labeled with the sum of the interaction energies of all rotamers on the path from the node to the root.

When the construction of a layer is completed, we sort the list of current leaves according to their energy labels. If the number of leaf nodes exceeds a certain bound `MAX_NODES`, we keep only the best `MAX_NODES` leaves and remove all others. As we keep always at most `MAX_NODES` nodes on each layer, we can be sure that the data structure does not grow exponentially.

Carrying out these steps for all $k$ side chains in the list $BS$ creates a layered tree of height $k$, with layer $i$ containing only rotamers of side chain $i$ (see Figure 3.3). The energy label of a leaf gives the potential energy resulting from interactions among the side chains and between the side chains and the template for the specific combination of rotamers defined by the rotamer labels on the path. We thus just have to pick the path from the leaf with the lowest energy label to the root in order to obtain the minimum energy conformation of all remaining conformations in the tree.

### 3.2.4 The Branch-&-Cut Algorithm

The GMEC problem can be formulated as follows: Given a list $BS$ of residue side chains and sets $V_i = \{i_1 \dots i_{n_i}\}$ of possible rotameric states for all $i \in BS$, determine for each side chain $i$ the rotamer $i_r$, such that the potential energy

$$E^{total} = E^{tpl} + \sum_i E^{tpl}_{i_r} + \sum_i \sum_{j<i} E^{pw}_{i_r,j_s}$$

of the selected conformation is minimized. We assume that all potential energies $E^{tpl}_{i_r}$ and $E^{pw}_{i_r,j_s}$ have already been computed. Let $E_{max}$ be the largest energy value among these potential energies.

We reformulate the problem as a minimization problem on an undirected graph $G = (V, E)$. For each rotamer $i_r$, we create a node $v$ with weight $E(v) = E^{tpl}_{i_r} - E_{max}$ and for each pair $(i_r, j_s)$ of rotamers with $i \neq j$, we create an edge $uv$ with weight $E(uv) = E^{pw}_{i_r,j_s} - E_{max}$. The resulting graph is $k$-partite with partitions $V_i$, $i = 1 \dots k$ where all the nodes and edges

have negative weights. The partition $V_i$ is called the $i$-th column of the graph. For a node $v$ of the graph, we define $c(v)$ to be the column of this node. The possible rotamer sets correspond to the subgraphs of $G$ with the following property: Every subgraph consists of exactly $k$ nodes, one node out of each column, and the induced edges. We call these subgraphs *rotamer graphs*. The weight of a rotamer graph is the sum of the weights of its nodes and edges. Note that the weight of a rotamer graph and the energy of its corresponding rotamer combination differ exactly by $E^{tpl} + (k + \binom{k}{2}) \cdot E_{max}$. Thus, the GMEC problem can be solved by determining the rotamer graph of $G$ with minimal weight.

### The Integer Linear Program

We now transform this graph-theoretic description into an integer linear program by introducing a binary decision variable $x_v$ for each node $v$ and $x_{uv}$ for each edge $uv$. If a node (edge) belongs to the rotamer graph, the value of the variable $x_v$ ($x_{uv}$) is 1 and otherwise it is 0. For brevity, we say that a node or an edge is selected if the corresponding binary decision variable is 1. The basic constraint system of the GMEC problem is the following:

$$\min \left( \sum_{v \in V} E(v) x_v + \sum_{uv \in E} E(uv) x_{uv} \right)$$

$$\begin{aligned}
s.t. \quad \sum_{v \in V_i} x_v = 1 \quad & \text{for all } i \in \{1 \dots k\} && (3.3)\\
x_{uv} \leq x_v \quad & \text{for all } uv \in E && (3.4)\\
x_{uv} \leq x_u \quad & \text{for all } uv \in E && (3.5)\\
x_v \in \{0,1\} \quad & \text{for all } v \in V && (3.6)\\
x_{uv} \in \{0,1\} \quad & \text{for all } uv \in E && (3.7)
\end{aligned}$$

The constraints (3.3) enforce that exactly one node of each partition, *i.e.* exactly one rotamer for each side chain, is selected. The constraints (3.4) and (3.5) guarantee that an edge can only be selected if both endpoints are selected, *i.e.* we include the pairwise interaction energy between two rotamers only if both rotamers are selected as well. Since only one edge from a certain column $i$ to a node $v$ can be selected, we can tighten these inequalities to

$$\sum_{u \in V_i} x_{uv} \leq x_v \quad \text{for all } v \in V, i \neq c(v). \tag{3.8}$$

Note that the above integer linear program has subgraphs of rotamer graphs as feasible solutions. Of course, the weight of a complete rotamer graph is smaller than the weight of any of its subgraphs, so that the optimal solution of the integer linear program is a rotamer graph.

### Branch-&-Cut

Branch-&-cut is the most common technique to handle hard combinatorial optimization problems. It works as follows: We relax the integer linear program by dropping the integrality condition and solve the resulting linear program. If the solution $\bar{x}$ is integral we have the optimal solution. Otherwise, we search for a valid inequality $fx \leq f_0$ that cuts off the solution $\bar{x}$,

*i.e.* $fy \leq f_0$ for all feasible solutions $y$ and $f\bar{x} > f_0$; the set $\{x \mid fx = f_0\}$ is called a cutting plane. The search for the cutting plane is called the separation problem. Any cutting plane found is added to the linear program and the linear program is resolved. The generation of cutting planes is repeated until either an optimal solution is found or the search for a cutting plane fails. In the second case a branch step follows: We generate two subproblems by setting one fractional variable $x_v$ ($x_{uv}$) to 0 in the first subproblem and to 1 in the second subproblem and solve these subproblems recursively. This gives rise to an enumeration tree of subproblems.

For details about branch-&-cut, integer programming, and a discussion of improvements on this method, see the book of Wolsey [131].

### The GMEC polyhedron

We call the convex hull $P$ of all feasible solutions the *GMEC polyhedron*. Since its facets are the most promising cutting planes, we studied the structure of the GMEC polyhedron. Before we can prove that an inequality defines a facet, we have to determine the dimension of the polyhedron. We assume that the reader is familiar with polyhedral theory; for an introduction, see [87, 131].

**Definition:** A *polyhedron* $P \subseteq \mathbf{R}^n$ is the set of points that satisfies a finite number of linear inequalities. Every polyhedron $P$ can be written as $P = \{x \in \mathbf{R}^n \mid Ax \leq d\}$ for some $m \times n$-matrix $A$ and some $m$-vector $d$. Note that this description of a polyhedron is not unique. For $i$, $1 \leq i \leq m$ let $A_i$ be the $i$-th row of $A$. If $A_i x = d_i$ for some $x \in \mathbf{R}^n$ the constraint $A_i x \leq d_i$ is called *active* for $x$. Let $M$ be the set of indices, so that $i \in M$ if and only if $A_i x \leq d_i$ is active for all $x \in P$. Furthermore, let $(A^=, d^=)$ be the corresponding rows of $(A, d)$. In other words, $A^= x \leq d^=$ are the constraints of $Ax \leq d$ which hold with equality for all points of the polyhedron.

**Lemma 1.** $\dim(P) = |V| - k + |E|$

*Proof.* A fundamental lemma about polyhedral theory (see, for example, [87], page 87) states that

$$dim(P) + rank(A^=, d^=) = |V| + |E|,$$

where $rank(A^=, d^=)$ is the number of linearly independent rows of $(A^=, d^=)$. Note that $rank(A^=, d^=)$ is independent of the particular description of $P$.

Hence, $\dim(P) \leq |V| - k + |E|$, because we have $k$ linearly independent equalities in our partial description of the polyhedron. In order to show that $\dim(P) \geq |V| - k + |E|$, we have to show that there are $|V| - k + |E| + 1$ affinely independent feasible solutions.

A *basic solution* $b$ is a feasible solution with $b_{uv} = 0$ for all $uv \in E$. For $i$, $1 \leq i \leq k$ we define $b(i)$ to be the node $v$ of $V_i$ with $b_v = 1$. For any node $v$ let $e_v$ be the unit vector of this node, *i.e.* the vector with entry 1 at the position of $v$ and with entry 0 at all other positions. Analogously for any edge $uv$, let $e_{uv}$ be the unit vector of this edge. For an $0/1$-vector $x$ we do not distinguish between the vector and the corresponding subgraph.

Let $b$ be a basic solution. We then construct all solutions where exactly one node of $b$ is replaced by another one in the same column. Additionally, we construct one solution for every

edge by including the edge, both its endpoints. For the remaining columns, we take the nodes of the basic solution $b$. More formally, we look at the following set $S$ of feasible solutions:

$$\begin{aligned} S = \{b\} &\cup \{b + e_v - e_{b(c(v))} \mid v \in V \text{ with } b_v = 0\} \\ &\cup \{b + e_{uv} + e_u - e_{b(c(u))} + e_v - e_{b(c(v))} \mid uv \in E\} \end{aligned}$$

The size of the set is $1 + |V| - k + |E|$. Let $M$ be the matrix with rows corresponding to the solutions of $S$. We have to show that the solutions are affinely independent, *i.e.* if one subtracts one row of $M$ from the others, the remaining rows are linear independent. In this case, the matrix has to have full row rank. We denote the $n \times n$ identity matrix by $I_n$. We subtract $b$ from all other solutions and get the matrix $M'$ with rows $e_v - e_{b(c(v))}$ for all $v \in V$ with $b_v = 0$ and rows $e_{uv} + e_u - e_{b(c(u))} + e_v - e_{b(c(v))}$ for all $uv \in E$. Since dropping columns cannot increase the row rank of the matrix, we are allowed to drop all columns $i$ of $M'$ with $b_i = 1$, *i.e.* all columns that have an entry $-1$. This results in a matrix with the following block-structure

$$\begin{pmatrix} I_{|V|-k} & 0 \\ B & I_{|E|} \end{pmatrix}$$

for some suitable matrix $B$. This matrix has full rank since it is triangular. Thus, the matrix $M'$ has full row rank and all solutions of $S$ are affinely independent.

□

**Lemma 2.** *The non-negativity constraints $x_{uv} \geq 0$ define facets for all $uv \in E$. The one-node-aggregation constraints (3.8) define facets for $P$.*

*Proof.* A feasible constraint defines a facet, if it is active for $|V| - k + |E|$ affinely independent solutions.

For the first part, we have to show that $x_{uv} = 0$ for $|V| - k + |E|$ affinely independent solutions. This is obvious, because all but one of the solutions in the proof of Lemma 1 satisfy this equality. The second part follows if we choose a basic solution $b$ with $b_v = 0$. Then $\sum_{u \in V_i} x_{uv} = x_v$ for all but one selected solutions.

□

**Definition:** Let $h, i, j$ be pairwise distinct column indices, $(S_1, S_2)$ be a partition of $V_i$, $(T_1, T_2)$ be a partition of $V_j$ and $(R_1, R_2)$ be a partition of $V_h$. Furthermore, all sets are assumed to be non-empty. We define the edge-incompatibility constraint for $S_1, S_2, T_1, T_2, R_1, R_2$ as

$$\sum_{u \in S_2, v \in T_1} x_{uv} + \sum_{v \in T_2, w \in R_1} x_{vw} + \sum_{w \in R_2, u \in S_1} x_{uw} \leq 1 \tag{3.9}$$

**Lemma 3.** *The edge-incompatibility constraints are feasible for $P$ and define facets.*

*Proof.* We first show the feasibility of the constraint. If any edge of the constraint is selected, no other edge of the constraint can be selected, because at least one of its endpoints cannot be selected (see Fig. 3.4).

In order to prove that the constraints are facet-defining, we use an indirect way to show that there are $|V| - k + |E|$ affinely independent solutions for which the constraint is active.

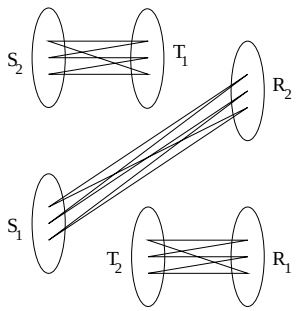



Figure 3.4: *Only one of these edges can be selected.*

Let $F = \{p^1, \ldots p^t\}$ be the set of feasible solutions for which the constraint (3.9) is active. The constraint defines a facet if all solutions $(\mu, \mu_0)$ of the set of equalities

$$\sum_{v \in V} \mu_v p_v^i + \sum_{uv \in E} \mu_{uv} p_{uv}^i = \mu_0 \text{ for all } p^i \in F \tag{3.10}$$

are linear combinations of the coefficients of the constraint (3.9) and the feasible equations $A^= x = b^=$ of the problem (see Theorem 3.6. in Chapter I of [87]), *i.e.*

$$(\mu, \mu_0) = (\alpha p^j + \mu A^=, \alpha + \mu b^=) \text{ for all } p^j \in F$$

The set of feasible equations consists of $k$ equations of the form

$$\sum_{v \in V_i} x_v = 1 \tag{3.11}$$

*i.e.*, one equation for each partition $V_i$. The scalar factor for the equation of column $V_i$ in the linear combination is denoted by $\lambda_i$, the scalar factor of the constraint (3.9) is denoted by $\alpha$. The above cited theorem implies that constraint (3.9) is facet-defining if

- $\mu_v = \lambda_i$ for each node $v \in V_i$ (because $p_v^j = 0$ for all $p^j \in F, v \in V$)
- $\mu_{uv} = \alpha$ for each edge $uv$ in constraint (3.9)
- $\mu_{uv} = 0$ for all other edges
- $\mu_0 = \alpha + \sum_{i=1}^{k} \lambda_i$.

Let $b$ be a basic solution. We call a constraint

$$\sum_{v \in V} \mu_v x_v + \sum_{uv \in E} \mu_{uv} x_{uv} = \mu_0 \tag{3.12}$$

*in normal form with respect to* $b$ if $\mu_v = 0$ for all $v$ with $b_v = 1$. By subtracting the equations (3.11) multiplied with suitable factors, it is possible to transform each constraint into (at least) one equivalent constraint in normal form with respect to $b$.

Thus, a constraint of type (3.9) defines a facet of $P$ if all solutions of the equation system

$$\begin{aligned} \sum_{v \in V} \mu_v p^i_v + \sum_{uv \in E} \mu_{uv} p^i_{uv} = \mu_0 \text{ for all } p^i \in F \\ \mu_v = 0 \text{ for all } v \text{ with } b_v = 1 \end{aligned} \tag{3.13}$$

are of the form

- $\mu_v = \lambda_i$ for each node $v \in V_i$
- $\mu_{uv} = \alpha$ for each edge $uv$ in constraint (3.9)
- $\mu_{uv} = 0$ for all other edges
- $\mu_0 = \alpha + \sum_{i=1}^{k} \lambda_i$.

Since every column $V_i$ contains one node $v$ with $\mu_v = 0$ (normal form), the above equations can only be fulfilled if all $\lambda_i$ are zero. Therefore, we have to show that $\mu_v = 0$ for each node $v \in V_i$, $\mu_{uv} = \alpha$ for each edge $uv$ in constraint (3.9), $\mu_{uv} = 0$ for all other edges, and $\mu_0 = \alpha$.

Choose any basic solution $b$, such that at least one edge $xy$ of the constraint is selectable, *i.e.* $b_x = 1$ and $b_y = 1$. Let $F$ be the set of all feasible solutions where exactly one edge of the constraint is chosen. We first argue that $\mu_v = 0$ for all nodes $v \in V$. By definition of the normal form, this is true for all nodes of the basic solution. Let $v \notin V_h \cup V_i \cup V_j$. Look at the following two solutions: the nodes of the basic solution together with the edge $xy$ and the solution where $v$ and $b(c(v))$ are exchanged and the same edge is selected. More formally, we look at the solutions $b + e_{xy}$ and $b + e_{xy} + e_v - e_{b(c(v))}$. Since $\mu_{b(c(v))} = 0$, the difference of the two resulting constraints is $\mu_v = 0$. Now let $v \in V_h \cup V_i \cup V_j$. Let $wr$ be an edge of the constraint that has neither endpoint in the same row as $v$. Take any solution where this edge and $b(c(v))$ are selected and the solution where $b(c(v))$ is replaced by $v$. Again the difference between the two constraints is $\mu_v = 0$.

Now we conclude that $\mu_{wr} = \mu_0$ for all edges $wr$ in the constraint (look at any solution where only this edge is selected), and that $\mu_{wr} = 0$ for all edges not in the constraint (look at any solution where this edge and one edge of the constraint is selected and the solution where only the edge of the constraint is selected). This proves that all solutions of the equation system (3.13) have the required form and we are done.

□

**Definition:** Let $h, i, j$ be pairwise distinct column indices, $S \subset V_i$, $T \subset V_j$ with $S, T \neq \emptyset$, and let $R_1, R_2$ be a partition of $V_h$ with $R_i \neq \emptyset$ for $i = 1, 2$. We define the node-aggregation constraints for $S, T, R_1, R_2$ as

$$\sum_{u \in S, v \in T} x_{uv} + \sum_{u \in S, w \in R_1} x_{uw} + \sum_{v \in T, w \in R_2} x_{vw} \leq \sum_{u \in S} x_u + \sum_{v \in T} x_v \tag{3.14}$$

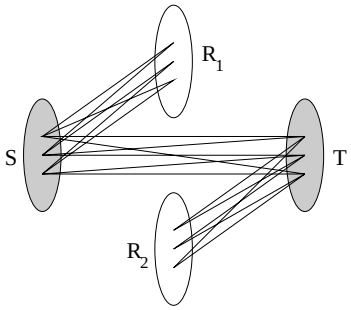



Figure 3.5: *One can select only as many edges as nodes in the shaded sets.*

**Lemma 4.** *The node-aggregation constraints (3.14) are feasible for $P$ and define facets.*

*Proof.* Again we start with the feasibility proof. If a node in $S$, a node in $T$, and a node in $R_1$ are selected we can select exactly one edge going from $S$ to $T$ and one edge going from $S$ to $R_1$ but no edges from $T$ to $R_2$. If a node in $S$ is selected but no node in $T$ we cannot select an edge from $S$ to $T$ or from $T$ to $R_2$. Furthermore, we can select at most one edge from $S$ to $R_1$. If neither a node from $S$ nor a node from $T$ is selected, we cannot select any of the edges of the constraint. The other cases are analogous (compare Fig. 3.5).

As in the proof of Lemma 3, we use Theorem 3.6. in Chapter I of [87] to show that the constraints are facet-defining. Let $F = \{p^1, \dots p^t\}$ be the set of feasible solutions that satisfy the constraint with equality. A constraint of type (3.14) defines a facet of $P$ if all solutions of the equation system

$$\sum_{v \in V} \mu_v p_v^i + \sum_{uv \in E} \mu_{uv} p_{uv}^i = \mu_0 \text{ for all } p^i \in F$$
$$\mu_v = 0 \text{ for all } v \text{ with } b_v = 1 \tag{3.15}$$

are linear combinations of the coefficients of the node-aggregation constraints and the feasible solutions with scalar factors $\alpha$ and $\lambda_i$. Thus, $(\mu, mu_0)$ has to be of the form

- $\mu_v = \lambda_i - \alpha$ for each node $v \in V_i, v \in S \cup T$ (because $p_v^j = -1$ for all $v \in S \cup T, p^j \in F$)
- $\mu_v = \lambda_i$ for each node $v \in V_i, v \notin S \cup T$
- $\mu_{uv} = \alpha$ for each edge $uv$ in constraint (3.14)
- $\mu_{uv} = 0$ for all other edges
- $\mu_0 = \alpha + \sum_{i=1}^{k} \lambda_i$.

Choose any basic solution $b$ with $b_u = 0$ for all $u \in S \cup T$. The constraint (3.14) is in normal form with respect to $b$. In a solution of this equation system, $\mu_v = 0$ for all $v \in V$ with $b_v = 1$.

Furthermore, $\mu_v = \lambda_i$ for all $v \in V$ with $b_v = 1$ by the choice of the basic solution and thus $\lambda_i = 0$ for all $1 \leq i \leq k$.

Therefore, we have to show that $\mu_v = -\alpha$ for each node $v \in S \cup T$, $\mu_v = 0$ for each node $v \notin S \cup T$, $\mu_{uv} = \alpha$ for each edge $uv$ in constraint (3.14), $\mu_{uv} = 0$ for all other edges, and $\mu_0 = \alpha$.

Let $w$ be the node in $V_h$ with $b_w = 1$ and w.l.o.g. assume $w \in R_1$ and $r$ is any node of $R_2$. Since the constraint is active for $b$ we conclude that $\mu_0 = 0$. Furthermore, we obtain:

1. $\mu_v = 0$ for all $v \in V$ that are not in the constraint (look at the difference between the constraints for the solutions $b$ and $b + e_v - e_{b(c(v))}$).

2. $\mu_{uv} = 0$ for all $uv \in E$ that are not in the constraint (look at the difference between the constraint of any active solution $p$ with $p_{uv} = 1$ and $p - e_{uv}$).

3. $\mu_u = -\mu_{uv}$ for all $u \in S$, $v \in R_1$ (look at the difference between the constraints of $b + e_v - e_{b(c(v))}$ and $b + e_v - e_{b(c(v))} + e_u - e_{b(c(u))} + e_{uv}$).

4. $\mu_u = -\mu_{uv}$ for all $u \in T$, $v \in R_2$ (The same argument).

5. $\mu_v = -\mu_{uv}$ for all $u \in S$, $v \in T$ (look at the difference between the constraints for the solutions $b + e_u - e_{b(c(u))} + e_{uw}$ and $b + e_u - e_{b(c(v))} + e_{uw} + e_v - e_{b(c(v))} + e_{uv}$).

6. $\mu_u = -\mu_{uv}$ for all $u \in S$, $v \in T$ (look at the difference between the constraints for the solutions $b + e_v - e_{b(c(v))} + e_r - e_w + e_{rv}$ and $b + e_v - e_{b(c(v))} + e_r - e_w + e_{rv} + e_u - e_{b(c(u))} + e_{uv}$).

Thus, for all $u \in S \cup T$ and $v \in V$ such that the edge $uv$ is in the constraint, we have shown that $\mu_u = -\mu_{uv}$. Since the subgraph induced by the constraint is connected, all $\mu_v$ have the same value. Choose $\alpha = -\mu_v$ for a node $v$ of the constraint. Then, $\mu_{uv} = -\mu_v = \alpha$ for each edge $uv$ of the constraint. Hence, all solutions of the equation system (3.13) have the required form and we are done.

□

We have implemented a branch-&-cut algorithm for the GMEC problem using the C++ class library LEDA [78], ABACUS [59] for the administration of the branch-&-cut tree, and CPLEX [52] to solve the linear programs.

We use the node-summation equations (3.3) and the one-node-aggregation constraints (3.8) as the initial constraint system. The only constraints that are separated are the node-aggregation constraints (3.14) with $|S| = |T| = 1$.

We use a simple, brute force algorithm. We iterate over all triples $(u, v, h)$, where $u, v \in V$, $h$ is a column of the graph $G$ and $c(u), c(v), h$ are pairwise distinct. The optimal partition is obtained by iterating over the nodes of $V_h$ and putting a node $w$ into $R_1$ if $x_{uw} > x_{vw}$ and into $R_2$ otherwise.

The running time is

$$O(\sum_{i=1}^{k}\sum_{j=1}^{k}\sum_{h=1}^{k}(|V_i||V_j||V_h|)) = O(|V|^3).$$

### 3.2.5 Energetic Evaluation

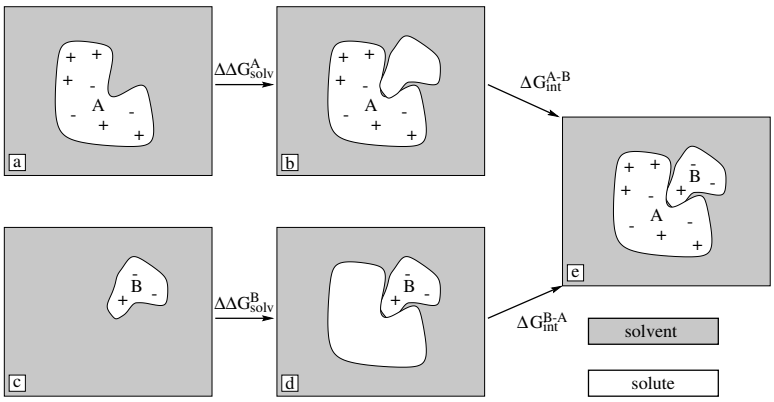



Figure 3.6: *Calculation of the electrostatic contributions to the binding free energy.*

After determining an optimal rotamer set for each candidate, we assign the rotamers of the minimum energy conformation to the respective side chains and then optimize the side chains of the binding-site using the AMBER force field. The free energy of binding is determined using the method proposed by Jackson and Sternberg [54].

We decompose the binding free energy $\Delta G_{ass}$ as follows:

$$\Delta G_{ass} = \Delta G^{ES} + \Delta G^{cav} + \Delta G^{conf} + \Delta G^{vdW} \tag{3.16}$$

$\Delta G^{ES}$ represents the electrostatic contribution, $\Delta G^{cav}$ the cavitation free energy in water, $\Delta G^{conf}$ the change in conformational entropy of the proteins, and $\Delta G^{vdW}$ the change in the van der Waals free energy.

#### Electrostatic contribution

The electrostatic contributions to the binding free energy can be estimated using a so-called *continuum model*. This type of model considers solvent molecules not explicitly, but implicit as a spatial dependency of the dielectric constant. The solute is represented by a region of low dielectric constant immersed in a high dielectric constant solvent (water). The boundary between these two regions is formed by the molecular surface of the solute (the solvent excluded surface).

Within this continuum model, the *Poisson-Boltzmann equation* (PBE) describes the electrostatic potential $\phi(\vec{r})$ at point $\vec{r}$ as a function of the charge distribution $\rho(\vec{r})$ and the (spatially varying) dielectric constant $\varepsilon(\vec{r})$:

$$\nabla(\varepsilon(\vec{r})\nabla\phi(\vec{r})) - \bar{\kappa}^2(\vec{r})\sinh\left(\frac{e^0\phi(\vec{r})}{kT}\right) = -\frac{\rho(\vec{r})}{\varepsilon_0} \tag{3.17}$$

The symbol $\nabla$ represents the gradient of a function. $\varepsilon_0$ is the vacuum permittivity and $e^0$ the proton charge. For our purposes, $\bar{\kappa}(\vec{r})$ (a modified Debye-Hückel parameter) may be set to zero without loss of accuracy:

$$\nabla(\varepsilon(\vec{r})\nabla\phi(\vec{r})) = -\frac{\rho(\vec{r})}{\varepsilon_0} \tag{3.18}$$

A common method to solve this equation is the finite difference method. After converting the linear PBE into its finite difference form, the resulting set of linear equations on a three-dimensional grid is usually solved using over-relaxation techniques [88]. The solution of the Poisson-Boltzmann equation yields the electrostatic potential for each grid point. The total electrostatic energy of the system may then be calculated as the sum of energies of each point charge $q_i$ in the electrostatic field:

$$\Delta G_{ele}^{total} = \frac{1}{2}\sum_i q_i\phi(\vec{r_i}) \tag{3.19}$$

In the Jackson-Sternberg model, the electrostatic contribution $\Delta G^{ES}$ to the binding free energy is composed as follows:

$$\Delta G^{ES} = \Delta\Delta G_{solv}^{A} + \Delta\Delta G_{solv}^{B} + \Delta G_{int}^{AB} \tag{3.20}$$

To evaluate the electrostatic contribution, we use the scheme in Fig. 3.6. First, we calculate the total electrostatic energy of $A$ in water (dielectric constant of water $\varepsilon_w = 80$, dielectric constant of the protein $\varepsilon_p = 2$). Then, we create a complex-shaped low dielectric constant cavity and charge only $A$. $B$ remains uncharged. The change in solvation free energy on binding $\Delta\Delta G_{solv}^{A}$ for $A$ is given by the difference between these two energies. The interaction energy of $B$ in the field of $A$ $\Delta G_{int}^{AB}$, which equals the interaction energy of $A$ in the field of $B$ $\Delta G_{int}^{BA}$, is then determined as the energy of the point charges of $B$ in the field caused by $A$. The same process is finally repeated for protein $B$ in the same manner. In our experiments, we used cubic grids with a spacing of $0.5$ Å and the Poisson-Boltzmann code implemented in BALL.

### Cavitation free energy

The change in cavitation free energy is calculated as a linear function in the change of the molecular surface area

$$\Delta G_{cav} = \gamma\,(A_{AB} - A_A - A_B), \tag{3.21}$$

where $A_{AB}$, $A_A$, and $A_B$ are the molecular surface areas of the complex, protein $A$, and protein $B$, respectively, and $\gamma$ is a constant ($0.289$ $\mathrm{kJmol^{-1}\AA^{-2}}$). We use the algorithm by Connolly [20] (as implemented in BALL) to calculate the molecular surface areas with a probe radius of $1.4$ Å. For the electrostatic calculations, we use the PARSE set of atom radii and charges by Sitkoff *et al.* [109] and the FDPB implementation of BALL.

## 3.3 Experimental Results

We applied our techniques to three protease-inhibitor complexes. The structures of three proteases $A$ and the corresponding protein inhibitors $B$ were taken from the PDB: subtilisin carlsberg (1SBC) with chymotrypsin inhibitor 2 (2CI2), $\beta$-trypsin (1TPO) with trypsin inhibitor (4PTI), and $\alpha$-chymotrypsin A (5CHA, chain A) with ovomucoid (2OVO). Water was removed from each of these complexes, hydrogens were added and their positions were optimized using the AMBER [22] force field.

### Combinatorial complexity

The iterative application of the DEE theorem reduced the combinatorial complexity usually by about ten orders of magnitude, thus greatly simplifying the solution of the GMEC problem. Numbers for the three examples are given in the following table:

| complex | no. of side chains | # rotamer combinations before DEE | # rotamer combinations after DEE |
|---|---|---|---|
| 1TPO/4PTI | 51 | $6 \cdot 10^{60}$ | $8 \cdot 10^{45}$ |
| 1SBC/2CI2 | 50 | $1 \cdot 10^{55}$ | $2 \cdot 10^{46}$ |
| 5CHA/2OVO | 54 | $1 \cdot 10^{56}$ | $2 \cdot 10^{48}$ |

### Ranking of the structures

In the case of 1TPO/4PTI, the AMBER energy already gave a clear signal; the best four candidates (those with lowest energy) were good approximations of the complex structure (see Fig. 3.7: structures are good approximations of the true complex structure if their RMSD[1] is low). When ranking this example with respect to the estimated binding free energy (calculated with the Jackson-Sternberg model as described in Section 3.2), an approximation of the true complex structure was ranked as number one as well (see Fig. 3.8).

For the example 1SBC/2CI2, the AMBER energy was not sufficient to predict the true complex structure correctly. However, when ranked according to the binding free energy, the best candidate was a good approximation of the true complex structure for all examples (see Fig. 3.9, 3.8, and 3.10).

We also tried to predict the structure of these examples using the binding free energy alone (*i.e.* without a prior side chain demangling). None of the complexes was correctly predicted.

[1] root mean square deviation of all atom position

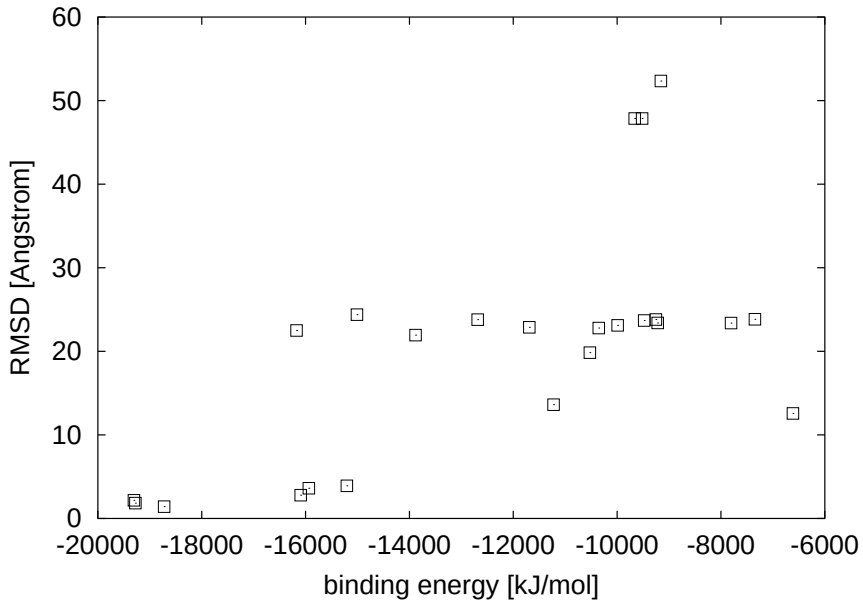



Figure 3.7: *1TPO/4PTI - AMBER energy of the candidates (MG).*

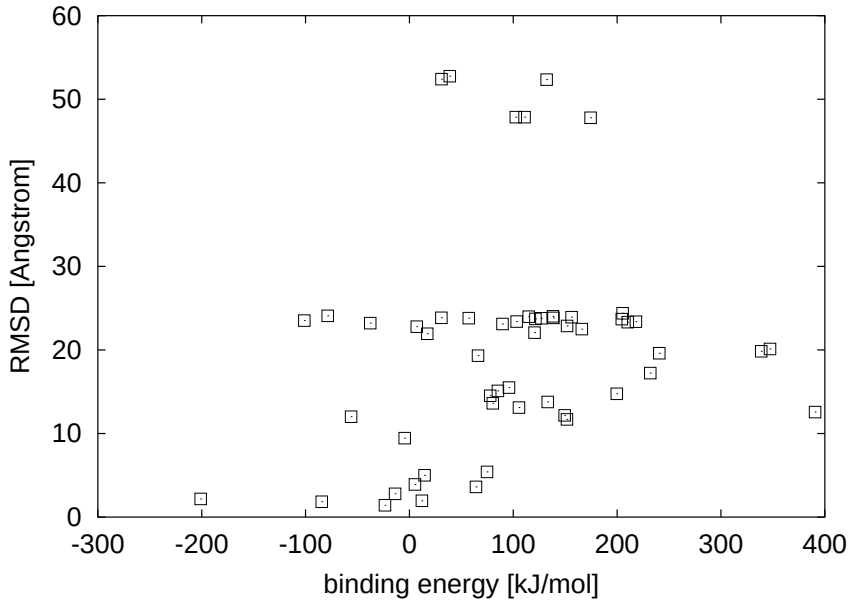



Figure 3.8: *1TPO/4PTI - ranking of the candidates according to their binding free energy (MG).*

Hence, side chain demangling as well as an advanced scoring method are both needed to predict the correct complex structure in these cases.

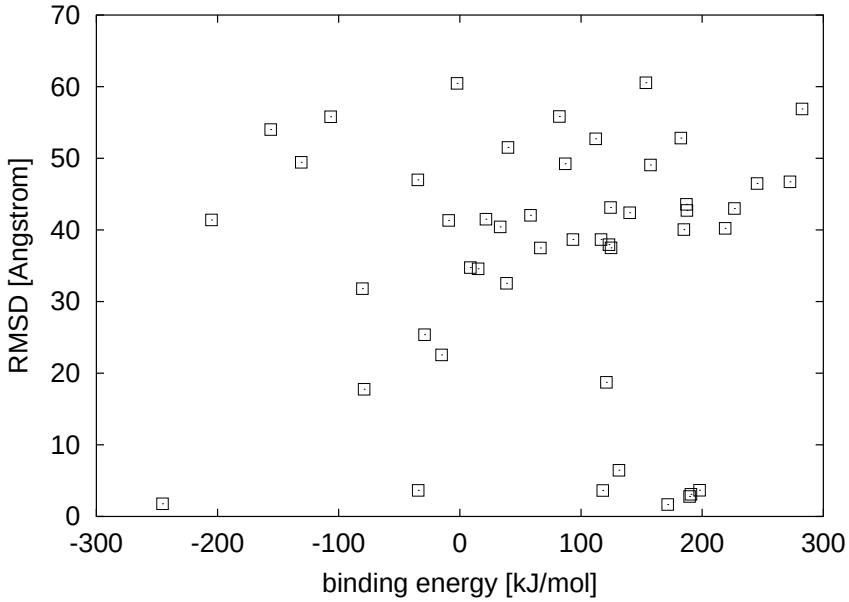



Figure 3.9: *1SBC/4CI2 - ranking of the candidates according to their binding free energy (MG).*

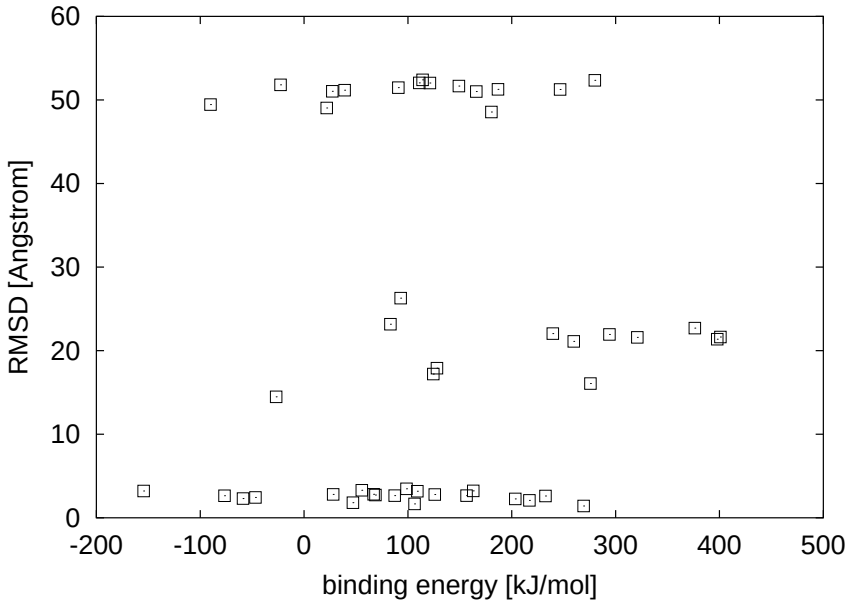



Figure 3.10: *5CHA/2OVO - ranking of the candidates according to their binding free energy (MG).*

### Quality of the heuristic

For the case of 1TPO/4PTI, we checked the quality of the results obtained from the MG method by calculating the optimal solution using the branch-&-cut algorithm. We performed this cal-

culation for 34 out of the 60 candidates. In most cases, the solution found by the heuristic was close to the optimal solution, *i.e.* for the majority of the side chains the selected rotamers were identical to those of the optimal solution. The total energy differed only slightly from that of the optimal solution, hence the ranking was very similar to that of the optimal solution (see Figs. 3.7 and 3.11). The average error in the AMBER energies was about $8000\ \mathrm{kJ/mol}$. However, the quality of the solutions was better for good approximations of the true complex structure. Among the 34 candidates considered, 7 were good approximations (RMSD $\leq 4$ Å). Only one of these seven candidates was further than $1000\ \mathrm{kJ/mol}$ away from the optimum.

### Side chain placement

For the protein docking problem, not all side chains are of equal importance. However, the placement of a single side chain may determine the success of the docking. A prominent example is LYS:15 of 4PTI. When compared to the bound structure, this side chain has to turn towards the core of 1TPO to fit properly. Fig. 3.12 shows the bound and the unbound conformation of this side chain. However, it is not necessary for LYS:15 to assume the fully extended position as in the complex structure, it just has to turn down a bit further than it does in the unbound structure to avoid clashes with the backbone of 1TPO. Although the correct placement of the side chain is not achieved, the selected rotamer is sufficient to allow a reliable prediction of the binding free energy.

### Running times

We compared the running times of the different parts of our docking algorithm on a SUN Enterprise 10000 (333 MHz UltraSparc II processors, 12 GB RAM, Solaris 7, g++-2.95.2 with -O2). We averaged the running times for each stage of the algorithm over all 60 candidates considered. Obviously, running time is dominated by the final energetic evaluation and the initial calculation of the energetic contributions, whereas the side chain placement itself only accounts for a minor portion of the total running time.

| stage | avg. time [min] |
|---|---|
| energies and DEE | 30 |
| ILP | 14 |
| multi-greedy | 3 |
| side chain optimization | 5 |
| final energetic evaluation (FDPB) | 70 |

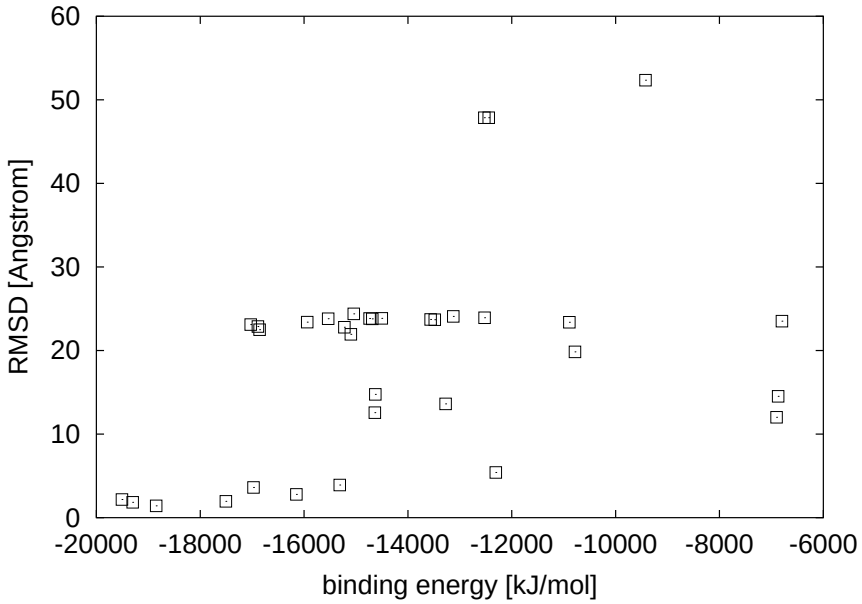



Figure 3.11: *1TPO/4PTI - AMBER energy of the candidates (ILP).*

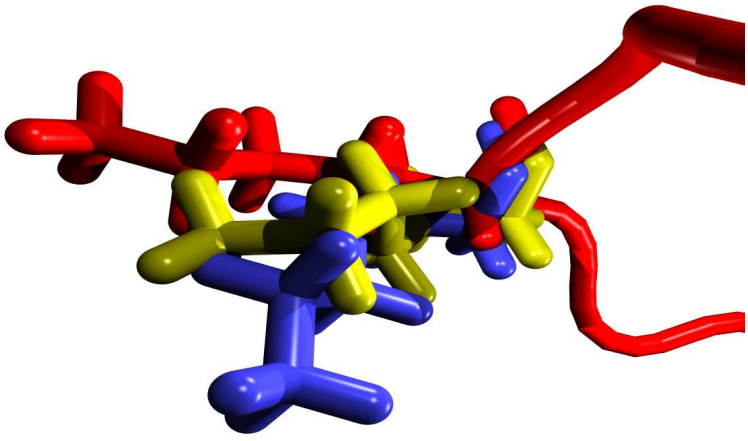


Figure 3.12: *Placement of LYS15 of 4PTI (red: complex conformation from 2PTC, blue: conformation from 4PTI, yellow: optimal rotamer).*

Chapter 4

# Protein Docking and NMR

## 4.1 Nuclear Magnetic Resonance Spectroscopy

One of the most important tasks in structural biology is the determination of the three-dimensional structure of biomolecules. Currently, there are three methods that are frequently employed in structure determination: *X-ray crystallography*, *Nuclear Magnetic Resonance* (NMR) spectroscopy, and *electron microscopy*. While the latter of the three methods is of minor importance and only yields low-resolution structures, X-ray crystallography and NMR spectroscopy are powerful tools to explore the structure of biomolecules at the atomic scale.

In X-ray crystallography, X-rays are focussed onto a protein crystal and the resulting diffraction patterns are recorded. From these patterns, the structure of the protein can be reconstructed through a *Fourier transform*. This technique is applicable to proteins of all sizes, the only prerequisite being the ability to grow crystals of the protein in question.

The second technique, which we will discuss in more depth, is the Nuclear Magnetic Resonance spectroscopy. The development of NMR spectroscopy was initiated by the pioneering work of F. Bloch and E. M. Purcell in 1945 [14, 101]. During the following decades, it became the method of choice for structure elucidation in organic chemistry. Due to new techniques and instruments, the size of the molecules accessible to NMR spectroscopy grew constantly until structure determination became feasible for small to medium sized proteins.

There is a whole range of textbooks covering the fundamentals and advanced topics of NMR spectroscopy (*e.g.* [48, 42, 120, 39, 45]), so we will just mention the fundamental physics and discuss the details only insofar as they are essential to the understanding of the subsequent sections.

### 4.1.1 The Nuclear Angular Momentum

NMR is based on the fact that the nuclei of most atoms possess a *nuclear angular momentum* $P$, which, according to the classical picture, corresponds to a rotation of the nucleus about an axis. From quantum mechanical considerations, one can deduce that the angular momentum of such an isolated particle cannot assume arbitrary magnitudes but is quantized (*i.e.* it may only take certain discrete values). For the angular momentum, these values can be specified in terms of a *quantum number* $I$:

$$|\vec{P}| = \sqrt{I(I+1)}\,\hbar, \tag{4.1}$$

where $h = 2\pi\hbar$ is the Planck constant. The *angular momentum quantum number* (or *spin quantum number*) $I$ is determined by the nucleus, *i.e.* it is determined by the isotope of the element (*nuclide*) in question. Nuclides with a spin quantum number of $I = 0$ do not possess an angular momentum and are thus not accessible through NMR spectroscopy (see Table 4.1).

| Nuclide | Natural abundance [%] | Spin quantum number $I$ | Gyromagnetic ratio $\gamma\,[10^7\mathrm{radT^{-1}s^{-1}}]$ |
|---|---|---|---|
| $^1$H | 99.985 | $\frac{1}{2}$ | 26.75 |
| $^2$H | 0.015 | 1 | 4.11 |
| $^{12}$C | 98.9 | 0 | – |
| $^{13}$C | 1.108 | $\frac{1}{2}$ | 6.73 |
| $^{14}$N | 99.63 | 1 | 1.93 |
| $^{15}$N | 0.37 | $\frac{1}{2}$ | −2.71 |
| $^{16}$O | 99.96 | 0 | – |
| $^{17}$O | 0.037 | $\frac{5}{2}$ | −3.63 |

Table 4.1: *Magnetic properties of the most important nuclides (values from [39]).*

Associated with the angular momentum $P$ is the *magnetic moment* $\mu$:

$$\vec{\mu} = \gamma \vec{P} \tag{4.2}$$

The *gyromagnetic ratio* $\gamma$ is a constant specific to each nuclide. Both $\vec{\mu}$ and $\vec{P}$ are vectorial properties, so their full description requires a direction as well. When placed in a static magnetic field $\vec{B}_0$ along the z-axis, the spins of the nuclei orient themselves along the field in a way that their z-component is quantized as well:

$$P_z = m\hbar \quad \text{with} \quad m = I, I-1, I-2, \ldots -I \tag{4.3}$$

where $m$ is the *magnetic quantum number*. Since $m$ can assume all values given in eq 4.3, $P_z$ (and $\vec{\mu}$) can assume $2I+1$ possibles values (see Fig. 4.1). Hence, the energy of the nucleus in the magnetic field $\vec{B}_0$ is quantized as well:

$$E = -\mu_z |\vec{B}_0| = -m\gamma\hbar|\vec{B}_0| \tag{4.4}$$

From here on, we will consider only nuclei with $I = \frac{1}{2}$ (*e.g.* $^1$H) for the sake of simplicity. Similar considerations apply to the other nuclei as well. $^1$H nuclei (*protons*), can assume two different energy levels in accordance with the two possible values of the magnetic quantum number $m = +\frac{1}{2}, -\frac{1}{2}$. If $\mu_z$ is positive, $\vec{\mu}$ is parallel to the external magnetic field, which is the preferred orientation. Transitions between these two states can be caused by the transfer of an appropriate amount of energy $\Delta E$. This energy has to correspond to the difference of the two energy levels, *i.e.*

$$\Delta E = E_{+\frac{1}{2}} - E_{-\frac{1}{2}} \overset{(4.4)}{=} \gamma\hbar|\vec{B}_0|. \tag{4.5}$$

The energies required for this transition are comparatively small. For typical NMR instruments, they correspond to electromagnetic radiation with frequencies in the range of several MHz to several hundreds of MHz. This frequency is called the *resonance frequency* of the nuclide.

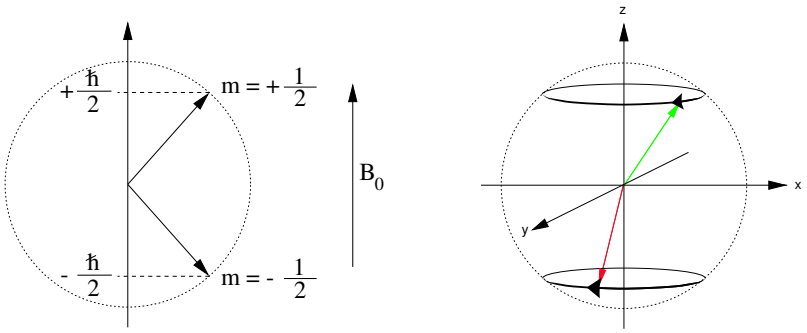



Figure 4.1: **Left:** *The nuclear angular momentum vector $\vec{P}$ of a proton ($^1H$) can assume two different states defined by its magnetic quantum number $m$. The z-component of the spin is either parallel or antiparallel to the external field $\vec{B}_0$.* **Right:** *Since this quantization condition holds only for the z-component, the spins may have arbitrary x- and z-components, i.e. they lie on cones defined by the total angular momentum $P$ and its z-component.*

According to the quantization condition (4.3), only the z-component is quantized. Classical physics require a magnetic moment in a field to perform a rotational movement around the z-axis which describes a cone (see Fig. 4.1). Such a movement is generally referred to as *precession* (analogous to the motion of a gyroscope). In the case of NMR, it is known as *Larmor precession*. The angular velocity $\omega$ of the precession and the corresponding *Larmor frequency* $\nu = \frac{\omega}{2\pi}$ are proportional to the external field $\vec{B}_0$:

$$\nu = \frac{|\Delta E|}{h} = \frac{|\gamma\hbar|}{h}|\vec{B}_0| \tag{4.6}$$

$$= \frac{|\gamma|}{2\pi}|\vec{B}_0| \tag{4.7}$$

$$\omega = |\gamma||\vec{B}_0| \tag{4.8}$$

### 4.1.2 Electronic Shielding and the Chemical Shift

From the equations stated above, one should expect all nuclei of a certain nuclide to show the same energy difference $\Delta E$ for a given magnetic field. However, atoms also contain electrons and these cause a local magnetic field which is superposed to the external field. As a result, the nucleus is shielded from the external field and experiences only an *effective field* $\vec{B}_{\text{eff}}$. The shielding is proportional to the external field and is therefore written in terms of the *shielding constant* $\sigma_{\alpha\beta}$:

$$\vec{B}_{\text{eff}} = \vec{B}_0 + \Delta\vec{B} = \vec{B}_0 - \sigma_{\alpha\beta}\vec{B}_0 \tag{4.9}$$

The shielding constant depends on the local environment (and thus on the molecular structure) of the nucleus in question. Since $\sigma_{\alpha\beta}$ is a tensor quantity, the effective field depends on the orientation of the molecule relative to the magnetic field vector $\vec{B}_0$. Hence, the random distribution of this orientation in liquid samples causes the non-isotropic contributions of $\sigma_{\alpha\beta}$ to vanish. The experiment only reveals the isotropic shielding constant $\sigma$ which is the trace of the

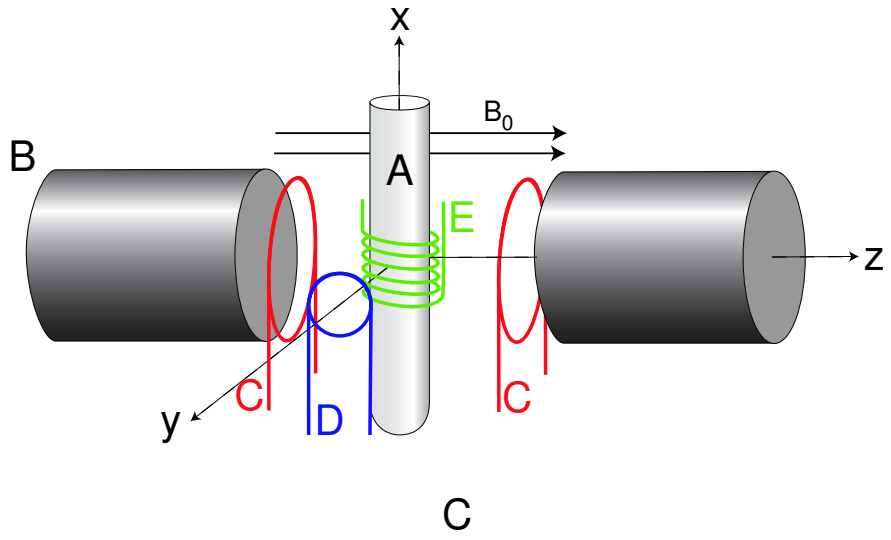



Figure 4.2: *Basic components of an NMR spectrometer: The sample (A) is placed in a strong homogeneous magnetic field $\vec{B}_0$ caused by a magnet (B). The field can be modified using the sweep coils (C). Radio pulses are transmitted using the transmitter coils (D) and received via the receiver coils (E).*

shielding tensor:

$$\sigma = \frac{1}{3}(\sigma_{xx} + \sigma_{yy} + \sigma_{zz}) \tag{4.10}$$

More practical than the shielding constant $\sigma$ is the chemical shifts scale (or $\delta$-scale). This scale describes the difference between the resonance frequencies of a given nucleus relative to the resonance frequency of a reference compound:

$$\delta = \frac{\nu - \nu_{\text{ref}}}{\nu_{\text{ref}}} \tag{4.11}$$

$\delta$ is named the *chemical shift* and is usually given in units of ppm (parts per million, $10^{-6}$). The chemical shift of the reference compound is zero by definition. Although the chemical shift is also a tensor quantity, we will regard it as a scalar quantity for the reasons stated above for the shielding tensor.

### 4.1.3 The Basic NMR Experiment

Fig. 4.2 shows the schematic construction of an NMR spectrometer. It consists of a strong (usually superconducting) magnet whose primary field $\vec{B}_0$ is modified by an additional field $\vec{B}_1$ created by a set of *sweep coils*. The sample is placed in this homogeneous field in a cylindric sample tube. A radio frequency of varying frequency is emitted by the *transmitter coil* and the resulting signal is picked up by the *receiver coils* surrounding the sample. A spectrum is acquired either by a *field sweep* (the current through the sweep coils is modified

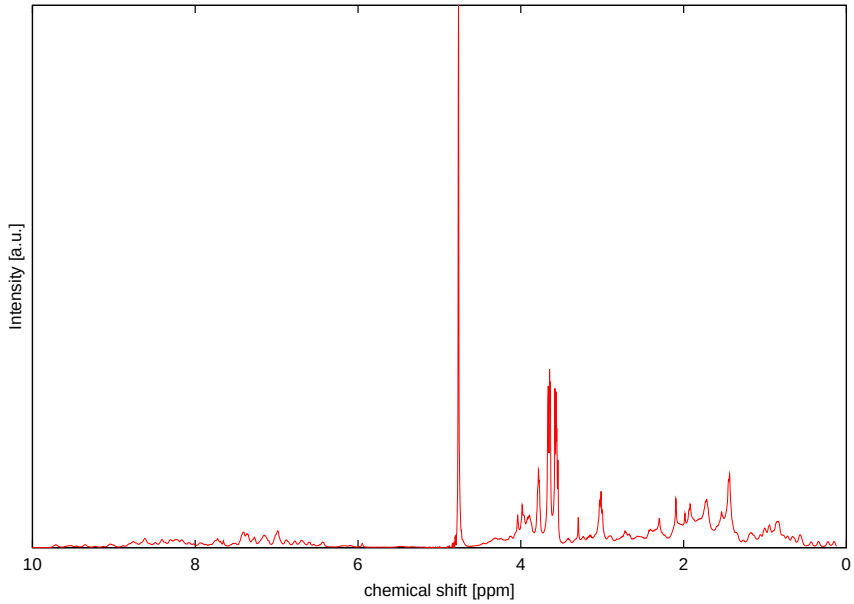



Figure 4.3: *The experimental $^1$H-NMR spectrum of the protein parvulin [7]. The large peak at $\delta = 4.7$ is caused by the solvent, the other peaks are caused by the protein itself.*

and the emitted radio frequency is kept constant), or by a *frequency sweep* (the field stays constant and the frequency varies). The intensity of the received signal is modulated by the varying absorption of the sample, which is at a maximum for the resonance frequency of each species in the sample. Thus, the spectrum shows a number of peaks corresponding to the chemical shifts of the different nuclei in the sample. The peak area is roughly proportional to the concentration of the species. In the case of proteins, the number of chemically distinct protons is very high and the spectra become very complex (Fig. 4.3).

There is a multitude of different experimental techniques available to perform an NMR experiment. Even a cursory overview of all available techniques is clearly beyond the scope of this work, so we refer to appropriate textbooks [48, 42, 120, 39, 45]. For our purposes, it is sufficient to know that the chemical shifts of the nuclei can be determined experimentally and that these shifts contain important structural information. Through different kinds of NMR experiments, it is possible to assign each of the peaks to the corresponding atom of the protein. This difficult task is called *shift assignment*. It requires a large number of different spectra, a lot of experience, and usually several weeks or even months of time.

A huge amount of structural data can be derived from a fully assigned spectrum. Depending on the kind of experiment that lead to the spectrum, this information consists of interatomic distances or secondary structure information. This information then allows the reconstruction of the 3D-structure of the protein.

## 4.2 Application to the Protein Docking Problem

As NMR spectra contain a huge amount of structural data and are easily accessible, it seems obvious to use this information in protein docking. The integration of experimental data into docking algorithms can improve the quality and reliability of docking results. Furthermore, the results of the docking predictions can be used to accelerate the structure elucidation process.

### 4.2.1 Previous Work

The inclusion of NMR data is a well known technique in ligand docking (*e.g.* [98]), but up to now, there is just one paper describing the use of NMR in protein-protein docking [82]. In that paper, Morelli *et al.* determine the changes in a two-dimensional ($^{1}$H-$^{15}$N-HSQC) spectrum upon the binding of ferredoxin to cytochrome $c_{553}$. From a set of (fully assigned) spectra, they determine a set of nuclei whose chemical shift changes on binding and thus obtain a set of distance constraints. An initial set of tentative complex structures is generated using a simple rigid-body docking algorithm. The best candidates are ranked with respect to the number of distance constraints they violate. Hence, the algorithm can be seen as a kind of local rigid-body docking, where the binding site and the coarse orientation of the two proteins is defined by the distance constraints. However, the NMR data is not directly exploited but only distance information derived from the fully assigned spectrum. To obtain this distance information, it is necessary to assign the majority of the shifts in the spectra (*e.g.* heteronuclear $^{1}$H-$^{15}$N-HSQC spectra).

Since the shift assignment is the most time-consuming process during the NMR-based structure elucidation, a more favorable approach would employ the unassigned spectra. In principle, it should be feasible to acquire an one- or multi-dimensional NMR spectrum of the complex and compare it to predicted spectra of the complex candidates. The simulated spectra of those candidates that are closest to the true complex structure should show the smallest deviation from the experimental spectrum. That leaves two open questions: (a) how to simulate a spectrum?, and (b) how to compare two spectra? The next sections are dedicated to these two problems.

### 4.2.2 NMR Shift Prediction

If we set aside spin coupling (an effect that splits peaks due to spin-spin interactions of neighboring nuclei), we can assign a chemical shift to each distinct nucleus of a protein. The chemical shift of a nucleus is influenced by numerous physical effects. Basically, all influences that lead to a change in the electronic structure of an atom (*i.e.* which influence the electron density) affect the chemical shift. Over the years, a number of individual effects have been described that are known to have an influence on the chemical shift, the most prominent ones being

- **ring currents**: in a simple classical model, the $\pi$-electrons of aromatic rings form current loops causing a magnetic field that superimposes the external field
- **electric field**: polar groups close to the nucleus can influence the shielding of a nucleus through electrostatic interactions

- **magnetic anisotropy**: the magnetic anisotropy of double bonds, especially the peptide group of proteins, can cause significant changes in the chemical shift of neighboring nuclei

- **protein secondary structure**: especially the heavier nuclei ($^{15}$N, $^{13}$C) display a strong dependency on the protein secondary structure

A number of common models have been proposed for these effects [92, 130, 129]. One of these models was proposed by Williamson and Asakura [129] (WA model). They decompose the total chemical shift of a nucleus into the following four contributions:

$$\delta = \delta_{\text{local}} + \delta_{\text{rc}} + \delta_{\text{aniso}} + \delta_{\text{es}} \tag{4.12}$$

where $\delta_{\text{rc}}$ is the contribution of ring currents, $\delta_{\text{aniso}}$ is the contribution caused by the magnetic anisotropy of the peptide bond, and $\delta_{\text{es}}$ is the electrostatic contribution. $\delta_{\text{local}}$ is the so-called local or *random coil shift* and equals the shift this nucleus had, if it were in a short peptide without secondary structure (random coil). For each of these effects, models were developed to estimate their contribution. The next paragraphs will describe these models in more detail.

### Ring Currents

The circular $\pi$-electron system of aromatic rings induces a magnetic field. This field changes the effective magnetic field at the nucleus and thus leads to a different chemical shift.

There are two widely used approaches to calculate the ring current shift: the approach by Haigh and Mallion [46] and the one by Johnson and Bovey [58]. For both models, the secondary shift can be written as

$$\delta_{\text{rc}} = \sum_{k \in \text{rings}} i_k \, B \, G(\vec{R}_k) \tag{4.13}$$

where $B$ is a constant, $i_k$ gives the ring current intensity of the $k$-th aromatic ring (relative to the intensity of a benzene ring), and $G(\vec{R}_k)$ is a geometric factor which depends on the shift model in use as well as on the position $\vec{R}_k$ of the $k$-th ring center relative to the proton.

In the Haigh-Mallion model the geometric factor is written as

$$G_{\text{HM}}(\vec{R}) = \sum_{i<j} S_{ij} \left( \frac{1}{r_i^3} + \frac{1}{r_j^3} \right) \tag{4.14}$$

where $r_i$ and $r_j$ are the distances of the nucleus from two neighboring atoms $i$ and $j$ of the aromatic ring, where the sum runs over all bonds of the ring. $S_{ij}$ is the area of the triangle spanned by the two ring atoms and the projection of the nucleus on the aromatic ring plane.

The Johnson-Bovey model is more complicated, but in general gives better accuracy. It is based on Pauling's model of the aromatic ring current [97] and fundamental ideas of Waugh and Fessenden [126]. Using Pauling's model, they estimate the ring current intensity in the

loop and calculate the magnetic field caused by the ring as that of two classical current-loops, one below and one above the aromatic ring plane. The resulting geometric factor is

$$G_{\mathrm{JB}}(\vec{R}) = \frac{1}{\sqrt{(a^2+\rho)^2+z^2}} \cdot \left( K + \frac{a^2-\rho^2-z^2}{(a-\rho)^2+z^2} E \right) \tag{4.15}$$

where $a$ is the ring radius, $\rho$ and $z$ give the position $\vec{R}$ of the nucleus relative to the ring center in cylindrical coordinates, and $K$ and $E$ are complete elliptic integrals of the first and second kind [110].

### Magnetic Anisotropy

The magnetic anisotropy is usually modeled by the approach of McConnell [76], which describes the contribution to the chemical shifts via the magnetic susceptibility tensor $\chi_{\alpha\beta}$ of the anisotropic group:

$$\delta_{\mathrm{aniso}} = \frac{1}{3N_A|\vec{R}|^3} \sum_{i=x,y,z} \chi_{ii}(3\cos^2\theta_i - 1) \tag{4.16}$$

Here, $\vec{R}$ is the distance vector from the nucleus to the anisotropic group, $N_A$ is the Avogadro constant, and $\theta_i$ is the angle between the $i$-axis and the vector $\vec{R}$.

### Electric Field

Electrostatic fields caused by polar groups can influence the chemical shift by deforming the nucleus' electron hull und thus changing the shielding. Buckingham [17] was the first to propose a formal treatment of the influence of an electrostatic field $\vec{E}$ on a C-H single bond by expanding the shielding tensor $\sigma_{\alpha\beta}$ as a power series:

$$\sigma_{\alpha\beta} = \sigma^{(0)}_{\alpha\beta} + \sum_{\gamma=x,y,z} \sigma^{(1)}_{\alpha\beta\gamma} E_\gamma + \frac{1}{2} \sum_{\gamma,\delta=x,y,z} \sigma^{(2)}_{\alpha\beta\gamma\delta} E_\gamma E_\delta + \ldots \tag{4.17}$$

This series is usually aborted after the quadratic term. The same expansion holds for the chemical shift tensor $\delta_{\alpha\beta}$ as well:

$$\delta_{\alpha\beta} = \delta^{(0)}_{\alpha\beta} + \sum_{\gamma=x,y,z} \delta^{(1)}_{\alpha\beta\gamma} E_\gamma + \frac{1}{2} \sum_{\gamma,\delta=x,y,z} \delta^{(2)}_{\alpha\beta\gamma\delta} E_\gamma E_\delta \tag{4.18}$$

Instead of the shielding tensor, we are usually interested in the isotropic chemical shift $\delta$. According to eq 4.10, $\delta$ is the trace of the chemical shift tensor $\delta_{\alpha\beta}$ and thus the above equation simplifies to

$$\delta = \delta_0 + \varepsilon_1 E_z + \varepsilon_2 |\vec{E}|^2 \tag{4.19}$$

where $\varepsilon_1$ and $\varepsilon_2$ are constants and $\vec{E}$ is the electric field at the bond with z-component $E_z$ (the z-axis coincides with the bond axis). $\delta_0$ is the chemical shift without an external electric field. We thus obtain for the electric field contribution alone:

$$\delta_{\text{es}} = \varepsilon_1 E_z + \varepsilon_2 |\vec{E}|^2 \tag{4.20}$$

Similar models have been proposed by a large number of authors (*e.g.* [92, 130, 129]). They usually differ only in the parameterization. These parameters are either derived from quantum mechanical calculations or they were fitted to experimental data.

| amino acid | atom name |
|---|---|
| all | N-terminal $NH_3^+$ protons |
| HIS | $H_{\delta 1}$, $H_{\varepsilon 2}$ |
| TYR | $H_\eta$ |
| SER | $H_\gamma$ |
| THR | $H_{\gamma 1}$ |
| CYS | $H_\gamma$ |
| ASN | $H_{\delta 2,1}$, $H_{\delta 2,2}$ |
| GLN | $H_{\varepsilon 2,1}$, $H_{\varepsilon 2,2}$ |
| LYS | $H_{\zeta 1}$, $H_{\zeta 2}$, $H_{\zeta 3}$ |
| ARG | $H_\varepsilon$, $H_{\eta 1,1}$, $H_{\eta 1,2}$, $H_{\eta 2,1}$, $H_{\eta 2,2}$ |

Table 4.2: *The protons that were assumed to be rapidly exchanging and thus invisible in the NMR spectra.*

### 4.2.3 Spectrum Synthesis and Comparison

After predicting the chemical shifts of all protons in a molecule, we have to reconstruct the NMR spectrum from this data. First, not all of these protons are seen in the spectrum. For example, hydroxyl, thiole, and some amine protons are usually not present. This is due to *intermolecular exchange*, *i.e.* the amino acid's protons are rapidly exchanged for protons of the solvent (water). This exchange happens on a faster time scale than the NMR experiment, so only a single average shift is seen for these protons and the solvent. Whether the proton is rapidly exchanged depends on the structure of the protein as well as on experimental conditions (*e.g.* pH of the sample) and is thus difficult to predict. Hence, we decided to use a list of protons that are known to exchange in the majority of all cases. Protons on this list were simply excluded from the shift list of the candidate and do not occur in the simulated spectrum. The list of excluded protons for our experiments is given in Table 4.2.

The refined shift list was then used to synthesize the spectrum. The peaks in an ideal NMR spectrum have *Lorentzian* line shape, *i.e.* the absorption intensity $A$ depends on the chemical shift via a function of the form

$$A(\delta) = \frac{1}{1 + \frac{(\delta - \delta_0)^2}{W}} \tag{4.21}$$

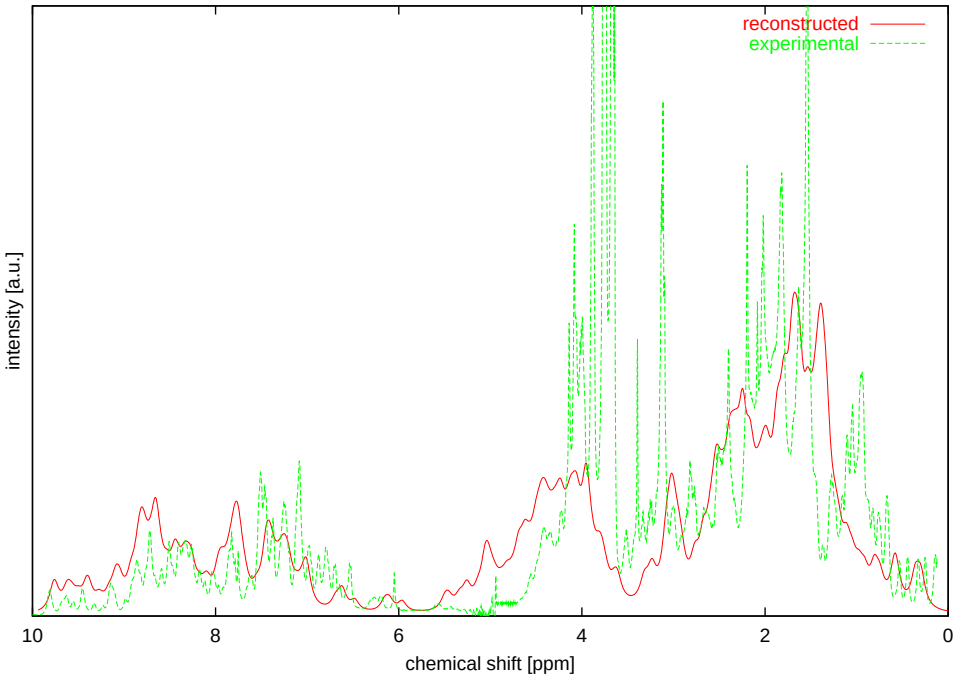



Figure 4.4: *Experimental and constructed $^1$H-NMR spectrum of the protein parvulin [7]. The solvent peak was removed.*

where $\delta_0$ is the chemical shift of the proton and $W$ describes the width of the peak. The line width is very difficult to predict, since it depends on a wide range of effects, ranging from the inhomogenity of the magnetic field to molecular motions. Therefore, we used a constant average line width for all protons in the spectrum. Then, we can write the spectrum $S$ as a linear combination of Lorentzians centered at the predicted shift positions $\delta_i$:

$$S(\delta) = \sum_i \frac{1}{1 + \frac{(\delta - \delta_i)^2}{W}} \tag{4.22}$$

For the case of parvulin, Figure 4.4 shows the experimental spectrum and the synthesized spectrum for comparison. Although there are significant differences between the two spectra, the overall structure is very similar. The differences are mainly caused by the differing line width in the experimental spectrum.

To compare two spectra, we calculated the unsigned difference between the two spectra. The integral of the resulting difference spectrum was estimated as the sum of 5000 equally spaced samples of the difference spectrum (in the shift range of -2 to 12 ppm). The absolute difference area of the two spectra $S_A$ and $S_B$ was calculated as the sum of all unsigned differences:

$$\Delta(S_\mathrm{A} - S_\mathrm{B}) = \sum_{\delta \in [-2,12]} |S_\mathrm{A}(\delta) - S_\mathrm{B}(\delta)| \tag{4.23}$$

The resulting difference area $\Delta$ was used as a measure of similarity between the two spectra and for ranking the docking results.

## 4.3 Experimental Results

In order to find a suitable test set for our approach we searched the contents of the BMRB (BioMagResBank [108]) for suitable protein complexes of known three-dimensional structure and available $^1$H-NMR spectra. Unfortunately, the number of candidates is very small. We identified four candidates where the BMRB contained rather complete shift data of the complex and a corresponding structure was deposited in the Protein Data Bank (PDB, [9]): the complex of calmodulin with the $Ca^{2+}$-calmodulin-dependent protein kinase kinase [94], the complex of calmodulin with a binding peptide of the $Ca^{2+}$-pump [30], the complex of S100B($\beta\beta$) with a peptide derived from p53 [104], and the two identical subunits of the homodimer S100B($\beta\beta$).

### 4.3.1 Methods

#### 4.3.1.1 Preparation of Structures and Rigid Body Docking

All complex structures were retrieved from the PDB (PDB IDs 1DT7, 1CFF, and 1CKK). From each structure containing several models, we selected the first model in the file. Missing hydrogens were added and all hydrogen positions were optimized in the complex using the AMBER 94 force field [22]. The complex structures were then separated into two files each containing one of the complexed proteins or peptides, which were used in our docking algorithm.

For each example, we carried out a rigid-body docking using the algorithm described in [70, 71]. The algorithm generates a list of tentative complex structures which are ranked with respect to geometric and energetic scoring functions.

#### 4.3.1.2 NMR Chemical Shift Calculation

Out shift model decomposes the total chemical shift $\delta$ of a proton into four components

$$\delta = \delta_{\mathrm{local}} + \delta_{\mathrm{JB}} + \delta_{\mathrm{aniso}} + \delta_{\mathrm{es}} \tag{4.24}$$

where $\delta_{\mathrm{local}}$ is the so-called random coil shift, $\delta_{\mathrm{aniso}}$ is the secondary shift caused by the magnetic anisotropy of the peptide bond, $\delta_{\mathrm{JB}}$ is the ring current effect as calculated by the Johnson-Bovey theory [47], and $\delta_{\mathrm{es}}$ is the effect of the electric field.

**Electric field effect** The electrostatic contribution was approximated using eq 4.20 with the parameters proposed by Williamson and Asakura [129] for both C-H and N-H bonds. The electric field was calculated via Coulomb's law with atomic charges taken from the AMBER 94 force field [22]. In contrast to our model, the WA model uses charges on the N, H, $C_\alpha$, C, and O atoms only. It has been reported in literature (*e.g.* [93]) that the use of force field charges yields slightly worse results than the use of charges on the backbone when predicting the spectra of proteins. A likely reason for this result is the fact that the charged side chains are often immersed in the solvent and the field arising from these charges is shielded by the solvent.

Since this shielding is not accounted for by the simple Coulomb model, the field caused by the side chain charges is overestimated.

In the case of protein docking however, we are not interested in the the spectrum alone, but in the differences of spectra. Since the main features of these difference spectra are governed by side chain–side chain contacts in the binding site, we decided to include charges for the side chains as well. As could have been expected, the use of the AMBER 94 charges gave better rankings than the use of backbone charges alone.

**Magnetic anisotropy** The magnetic anisotropy of the peptide group was modeled using McConnell's equation (eq 4.16). Again, we used the parameters proposed by Williamson and Asakura for the C=O and C-N bond of the peptide group.

**Ring current** We used the Johnson-Bovey model [58] which proved to give slightly better results than the Haigh-Mallion model. We used radii of 1.182 Å for the five-membered rings (HIS, TRP) and 1.39 Å for the six-membered rings (PHE, TYR, TRP).

**Random coil shift** Using a training set of 14 proteins with known structure and assigned $^1$H chemical shifts (obtained from the PDB [9] and the BMRB [108]), we fitted the random coil shifts to reproduce the experimental shifts as closely as possible. We used a test set of seven fully assigned protein structures as a test set to verify the improvement of the shift model. In fact, the standard deviation of all $^1$H chemical shifts in the test set was reduced from the initial 0.52 ppm obtained with random coil shifts from the BMRB to 0.44 ppm.

#### 4.3.1.3 Spectrum Synthesis and Comparison

**Spectrum synthesis from experimental data** The assigned chemical shifts were read from the BMRB files (BMRB IDs 4099, 4284, and 4270). The spectrum was then simulated by assuming a Lorentzian line shape of equal width for each proton (eq 4.21). We chose an average value of $W = 0.0032$ ppm$^2$.

**Spectrum synthesis from candidate structures** For each proton of the tentative complex structures, the chemical shift was calculated according to eq 4.24. Then, we removed the most exchangeable protons (see Table 4.2). We thus obtained a list of "observable" protons, which was used to create a spectrum as described above.

**Comparison** For comparison, we sampled the "experimental" spectrum $S_{\mathrm{exp}}$ and the spectrum of each tentative complex structure $S_{\mathrm{cpx}}$ in the range between -2 and +12 ppm at a total of 5000 regularly distributed positions (see eq 4.23). The resulting difference areas $\Delta$ were normalized by subtracting the smallest occurring area from all other areas. These values were then used to rank the structures.

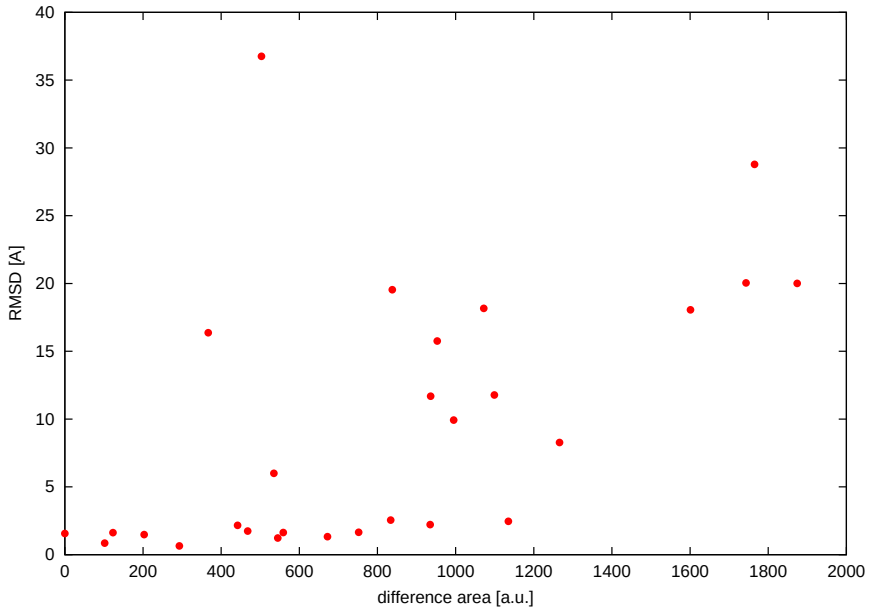



Figure 4.5: *Results of the docking of the S100B($\beta\beta$) dimer.*

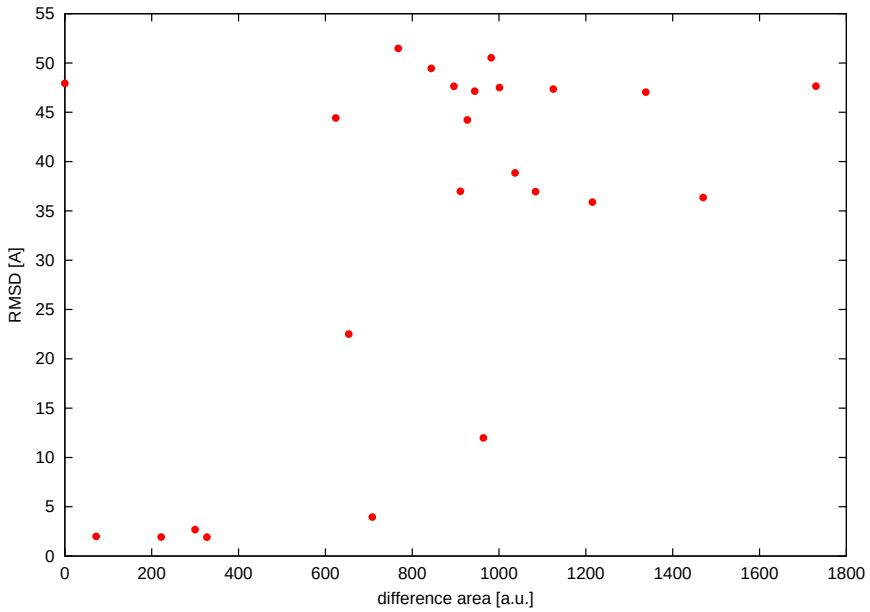



Figure 4.6: *Results of the docking of calmodulin with the binding peptide of the $Ca^{2+}$-pump.*

### 4.3.2 Results

The rigid docking of the four test cases resulted in four sets of tentative complex structures (each set with 24 to 121 different structures). For each of the potential complex structures, we

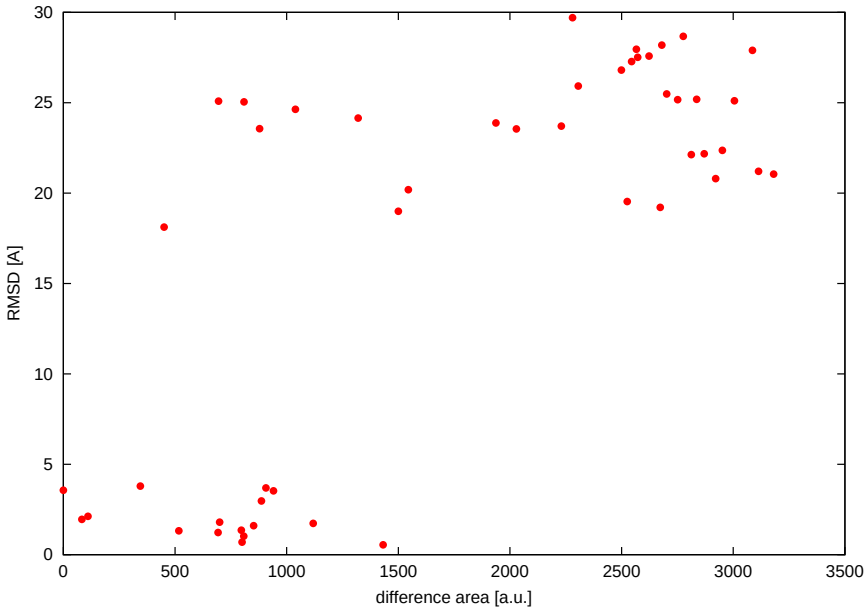



Figure 4.7: *Results of the docking of calmodulin with the $Ca^{2+}$-calmodulin-dependent protein kinase kinase.*

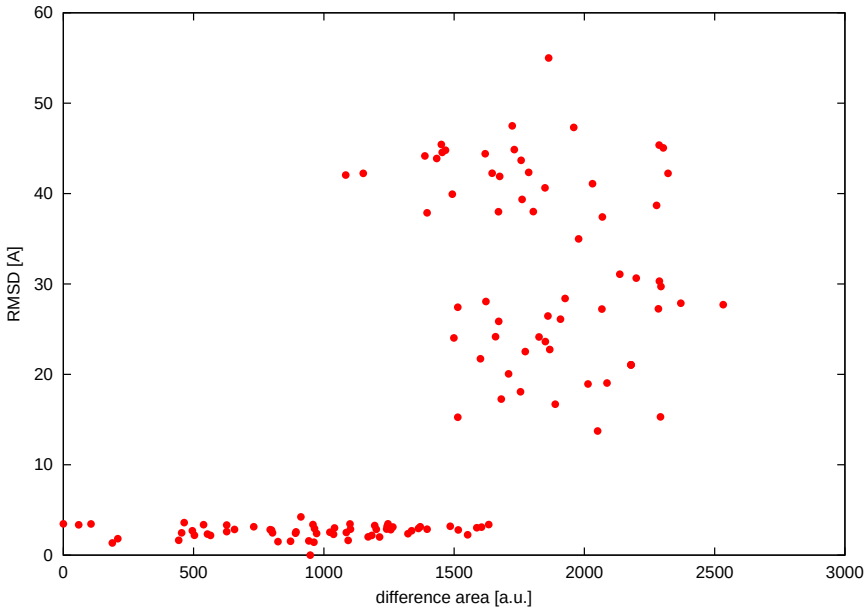



Figure 4.8: *Results of the docking of the S100B($\beta\beta$) dimer with a peptide derived from p53.*

calculated the $^1$H-NMR spectra and determined the difference area between this spectrum and the experimental complex spectrum. In the case of S100B($\beta\beta$), the BMRB did not contain the complete shift data of the peptide in complex with the S100($\beta\beta$) homodimer, but only the shifts of the homodimer itself.

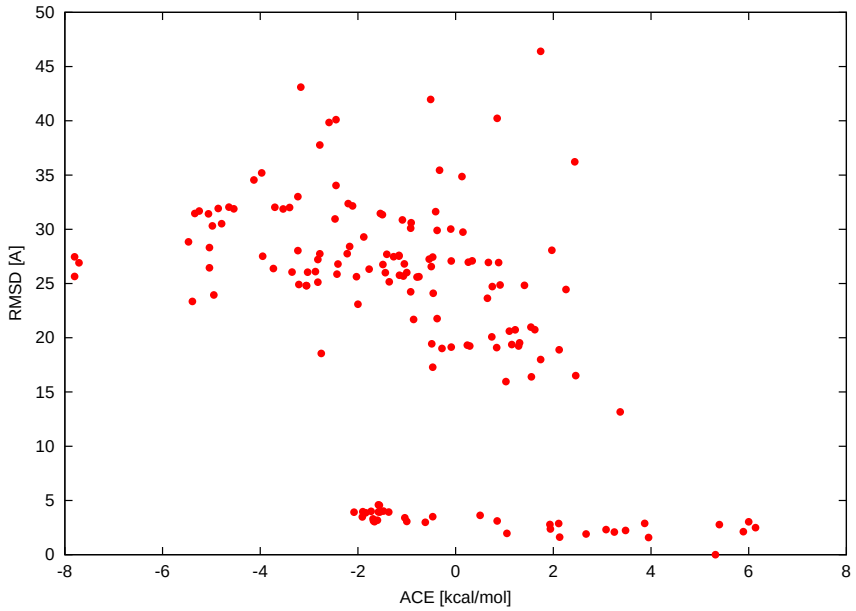



Figure 4.9: *Results of the conventional docking of the S100B($\beta\beta$) dimer with a peptide derived from p53. No NMR data was used. Instead, we employed the Atomic Contact Energy (ACE) by Zhang et al. [132] as a scoring function.*

Initial experiments showed that the ranking of structures was significantly improved, if the contributions stemming from the magnetic anisotropy of the peptide group were not included. This result seems surprising at first sight, but a detailed analysis of the shifts lead to the conclusion, that the effect is caused by structural overlaps. Slightly overlapping structures are a typical result of rigid-body docking algorithms. Even small deviations of the true complex structure can bring individual protons into a closer spatial vicinity to anisotropic groups than could be expected from the atoms' van-der-Waals radii. Since the effect of the magnetic anisotropy grows with the third power of the inverse distance, these collisions lead to enormous changes in the chemical shift. Furthermore, the effect of the magnetic anisotropy is basically a local effect; it depends strongly on the backbone torsion angles (*i.e.* the secondary structure) and has a much more limited range than ring current and electric field effects. Therefore, the magnetic anisotropy of the backbone of one of the proteins should not influence the shifts of its docking partner significantly. Hence, we excluded the effect of the magnetic anisotropy between the two docking partners, but included it within each of the partners.

The results of the docking experiments are shown in Figs. 4.5, 4.6, 4.7, and 4.8. In these figures, every point represents a single tentative complex structure. It shows the root mean square deviation (RMSD, y-axis) of the structure from the true complex structure and the normalized difference area of the candidate's spectrum (x-axis). Good approximations of the true complex structure should thus be expected in the lower left corner of the graph.

Except for the complex of calmodulin with the binding peptide of the $Ca^{2+}$-pump, scoring according to the difference area always identified a good approximation of the true complex structure. The separation between true and false positives was good for the S100B($\beta\beta$) dimer

and for the complex of calmodulin and kinase, and excellent for the complex of S100B($\beta\beta$) and the p53-peptide.

The latter fact is very surprising, since the docking of the small p53-derived peptide (22 amino acids) was impossible using conventional methods. We tested different energy-based scoring functions but we were not able to obtain a correct ranking. Fig. 4.9 shows the result of the docking using the Atomic Contact Energy (ACE) developed by Zhang *et al.* [132]. The first approximation of the true complex structure is ranked as number 48. Other scoring functions, *e.g.* the use of geometric methods [60] or the inclusion of electrostatics, gave very similar results. The problems with this docking example stem basically from the small binding site of the peptide. Most docking algorithms favor structures where the peptide has a larger contact area with the protein. In this case, the use of NMR data was the only possibility to correctly predict the complex structure.

For the complex of calmodulin with the binding peptide of the $Ca^{2+}$-pump, a false positive structure was ranked number one, followed by the major part of the true positive structures. The reasons for this failure are not yet clear.

Chapter 5

# Discussion

## Semi-flexible docking

We proposed two new approaches for semi-flexible protein docking. In a first step, a rigid-body docking algorithm generates a set of tentative complex structures. In a second step, a side chain demangling step, we resolve problems arising from side chain overlaps in the binding site. This demangling results in physically meaningful complex conformations that can be ranked with standard energetic functions.

The side chain demangling is based upon a reduction of the side chain placement problem to a combinatorial optimization problem by discretizing the side chain conformations to rotamers. The first approach is based on a simple greedy heuristic. It searches for minimal solutions in the enumeration tree spanned by the rotamers of all side chains in the binding site and avoids combinatorial explosion by limiting the number of leaves in each tree layer. This approach is very fast, although it usually yields suboptimal solutions. The quality of these solutions is nevertheless sufficient to solve the side chain placement problem in protein docking.

The second approach is a branch-&-cut technique that is based on an ILP formulation of the side chain placement problem. After identifying several classes of facet-defining inequalities for the side chain placement polytope and devising a separation algorithms for a subclass of these inequalities, we were able to solve the side chain placement problem to optimality.

Both approaches correctly predict the structure of three difficult test cases for protein docking that cannot be solved using rigid-body docking approaches. Running times are moderate and mainly governed by the energetic evaluation of the final structures and not by the side chain placement step.

There is still room for significant improvements of the method. The fact that we can solve the problem optimally unfortunately does not imply that we can solve the *biological problem* optimally. Instead, we solve an abstract representation of problem. There are two main differences between the two problems. First, we find an optimal solution with respect to an energetic function (the force field). This function is only a coarse approximation of the true energies occurring in the protein. Second, the restriction of the side chain conformational space to the rotamer conformations possibly exclude some rare conformations that may occur in nature. Both points leave much room for improvements. For example, the AMBER force field does not account for solvation effects. In contrast, the recently developed CHARMM EEF1 force field [66] introduces an additional force field term for the solvation energy. Since the solvation energy is expressed as a pairwise interaction, our energetic decomposition still holds for this force field and it might significantly improve the results. The rotamer library we used could also be replaced by a more detailed library, thus representing also less frequently occurring conformations.

Besides these improvements of the fundamental model, algorithmic and implementation improvements are possible as well. An interesting approach may be the use of branch-&-

price techniques. Also the most time-consuming step for the side chain placement problem (the calculation of the interaction energies) could be sped up by integrating the calculation into the greedy algorithm and thus calculating only those energies that are required, instead of precomputing all interactions in an initial step.

The key component of the algorithm, the side chain placement algorithm, can be applied to related problems like protein structure prediction or ligand docking as well.

### NMR-based docking

We also presented a new approach to protein docking that directly incorporates experimental structural data of the protein complex into the docking algorithm. It is the first algorithm that permits the use of NMR spectra for the validation and ranking of docking results. NMR-derived distance constraints have been widely used in ligand docking to improve the results, but these constraints have to be determined manually from fully assigned NMR spectra. Our method avoids this difficult and time-consuming manual preprocessing of the experimental data. Instead, we use the structures proposed by the docking algorithm to simulate the NMR spectrum of the complex and compare this spectrum directly to the experimental spectrum.

We implemented and tested several empirical models for NMR shift prediction. The model we used in our final experiments was based on existing models, that were combined and reparameterized. We also had to adapt the contributions arising from the magnetic anisotropy of the peptide bond and the electric field effect to the special requirements of the docking problem. The scoring function used for the docking is based on the difference area between the predicted and the experimental spectrum.

We chose a test set of four protein complexes: the complex of calmodulin with the $Ca^{2+}$-calmodulin-dependent protein kinase kinase, the complex of calmodulin with a binding peptide of the $Ca^{2+}$-pump, the complex of S100B($\beta\beta$) with a peptide derived from p53, and the two identical subunits of the homodimer S100B($\beta\beta$). For three of these complexes, our algorithm could clearly identify the true complex structure. in the case of the fourth complex (calmodulin with the binding peptide of the $Ca^{2+}$-pump) one false positive structure was ranked slightly better than the first true positive structure. The usefulness of the method has been proven for the complex of S100B($\beta\beta$) and the p53-peptide. We tried to predict the structure of the complex using several energy-based scoring functions, but were unable to predict the correct structure. When using our NMR-based scoring function, we obtained an excellent separation of true positives and false positives: there was no false positive among the first 20 candidates. Obviously, the method is a useful tool even in those cases, where energy-based methods fail.

One major problem when testing the algorithm was clearly the amount of data available. Since experimental spectra are not available from public data banks, we were forced to reconstruct the spectra from the deposited shift assignments and even then, there were only four test cases available. The main reason for the scarcity of data is the fact that X-ray crystallography is still the more common method for structure elucidation of protein-protein complexes. So the approach has still to be validated with direct data, which we hope to obtain from a collaboration with NMR spectroscopists.

We are also positive that the shift model proposed here can still be significantly improved. We carried out a series of quantum mechanical calculations to verify and reparameterize some

of the shift contributions. In the course of these experiments, we found that most models are quite coarse approximations. This observation is in accordance with recent developments in the field of NMR shift modeling, since many researchers in this field are currently working on quantum mechanical calculation of so-called shielding hypersurfaces. The current model also considers only a subset of all known effects. A detailed analysis of those shifts that showed the largest deviations between predicted and experimental shift values lead to the conclusion that hydrogen bonds and solvent effects play an important role. Models for both effects still have to be developed.

Further improvements should also address the prediction of the peak widths. The line width is caused (at least in parts) by the molecular motion of the protein. Therefore, it seems plausible to obtain better results by calculating the spectra as a time average over trajectories obtained from Molecular Dynamics Simulations. An interesting question is also whether this approach can be extended to ligand docking as well or whether the changes in the spectrum caused by small ligands are too insignificant in comparison with the protein spectrum.

Finally, and perhaps the most interesting application, is the use of the new scoring function in protein structure prediction. In protein structure prediction, the three-dimensional structure of a protein is predicted from its sequence alone. Similar to docking algorithms, protein structure prediction methods generate a set of potential structures, which are then ranked with respect to some energy function. The fact that the spectra-based scoring function gave very promising results for protein docking raises hopes that similar results can be obtained for protein structure prediction as well. These hopes are based on the fact that structural differences between the candidates obtained from structure prediction algorithms are larger than the differences between the candidates generated for protein docking. The combination of protein structure prediction and NMR spectra prediction could then speed up the process of structure elucidation for proteins significantly by generating good initial structure models and initial guesses for the shift assignment. Eventually, such methods could pave the way to high-throughput methods for protein structure elucidation, one of the foremost goals of proteomics.

# Part III

# BALL

Chapter 6

# Design and Implementation

## 6.1 Introduction

Implementation is often the biggest hurdle when testing new ideas and approaches. This is especially true for the field of Molecular Modeling, where the implementation of standard techniques (*e.g.* Molecular Mechanics) often requires several man years of hard work to implement. Obviously, this problem can be tackled using software that was specially designed for *Rapid Application Development* (RAD, also called *Rapid Software Prototyping*).

The need for appropriate RAD tools becomes even more obvious when looking at the software that is currently used. The majority of software packages currently in use in Molecular Modeling is not only written in FORTRAN, but often lacks a thorough documentation as well. The language FORTRAN itself is not very suitable for Rapid Software Prototyping, as the procedural programming paradigm is clearly inferior to the Object-Oriented approach with respect to reusability. Nevertheless, FORTRAN packages are still immensely successful, because people got used to their, often awkward, interfaces and people prefer the time a reimplementation would require preferably for other tasks.

We already discussed a number of software packages that are frequently used for Molecular Modeling in the introduction (Part I). To summarize this discussion, one can say that these packages provide the required functionality, but they are hardly reusable and (apart from some exceptions) not object-oriented. There is no software available that is specifically designed for Rapid Application Development. For this reason, we designed and implemented BALL – the Biochemical Algorithms Library.

When we started the BALL project (under the tentative project title BioLEDA at that time), we did not spend much time on design. In fact, we started in June 1996 with just one page describing the attributes of nine classes (`System`, `Molecule`, `Fragment`, `Atom`, `Sequence`, `BioMolecule`, `RNA`, `DNA`, and `Protein`). Based on this kernel of primitive classes, we wanted to implement algorithms for protein-protein docking.

A vast amount of code was implemented at the beginning of the project. It was not until we first wanted to use the code in real applications, that we realized it was completely unusable, although it fulfilled all our “specifications”. Thus, we learned the need for a thorough design the hard way. The current version of BALL was designed more carefully and we also made use of some more advanced software engineering techniques.

BALL was first designed to be a *class library*, but it soon turned out that instead we needed a *framework*. Contrary to a typical class library, the classes of a framework also provide abstract or empty methods. The framework defines the cooperation of its classes and the developer usually just fills in the required functionality by defining new subclasses. These subclasses redefine the abstract or empty methods of their respective base classes. Due to inheritance, the cooperation of the subclasses is still defined by the framework.

Another important technique is *generic programming*. Generic programming means "the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously" [84]. Generic programming became an important technique in the development of BALL. Especially the kernel strongly relies on generic data structures (*e.g.* geometric objects, lists, hash data structures, *etc.*). Furthermore, careful generic implementations of algorithms are very efficient as they avoid the overhead that is often implied by inheritance (which can replace generic programming in many cases).

Finally, we tried to use *design patterns* [41] as often as possible. Design patterns are intended to improve the object-oriented design through reuse at a very high level: the reuse of design solutions. It is often very hard to come to the right design decisions and to find the right balance between generality and specificity in the design of a class hierarchy. Design patterns are standard solutions to recurring design problems which have shown their usefulness in various applications. Hence, we tried to use design patterns wherever applicable to obtain a high quality design.

We will describe the design and the implementation of BALL in Chapter 6 and will then give a cursory overview of the techniques employed to ensure a sufficient code quality and to manage the project in Chapter 7.

BALL is intended as a framework for Rapid Software Prototyping in Molecular Modeling. With this goal in mind, we rethought our earlier approaches and performed a thorough analysis of existing software packages in this field (see also the discussion of existing packages in Part I on Page 5). This analysis led to the formulation of four major design goals. The next section will briefly discuss these goals. Our approaches to achieve these goals will then be discussed in the following sections and in Chapter 7.

## 6.2 Design Goals

### 6.2.1 Ease of Use

Ease of use is crucial for the acceptance of a new tool, especially if this tool claims to be designed for Rapid Software Prototyping. First of all, the user should be able to use the software rather intuitively – as far as this is possible for a framework of this size. It should therefore employ a standardized and widely used *programming language*. Furthermore, it should provide an intuitive, well documented, and consistent interface. Here, consistency means that the user can generalize anything he learned to all parts of the framework. For example, naming conventions should be globally applicable and functions with the same name should have the same or comparable effects.

Ease of use also implies a simple and smooth installation on all supported platforms, especially as most of our intended users are chemists or biologists that are usually not too familiar with the problems and pitfalls of installation on different platforms. Hence, this goal also implies a high portability.

In this context, we formulated what we called the *Three Line Rule*: it should be possible to code the majority of frequently occurring operations in not more than three lines of code.

### 6.2.2 Functionality

Functionality is of obvious importance. We can only expect wide acceptance of our software, if it is able to save the potential user a major amount of time. Thus, it should provide most of the standard functionality and leave only the implementation of specialized or new techniques to the user. And even there the user should be able to take advantage of existing data structures to speed up the development.

This is a very ambitious aim, because Molecular Modeling and Computational Molecular Biology are rather large and diverse fields and it seems impossible to provide all functionality anybody might require. So we had to put a focus on the more elementary things that are required by the majority of applications and we restricted the functionality to the field of Molecular Modeling and especially protein docking.

We identified four key areas of functionality that are used in most applications. First of all, we need some *molecular data structures* to represent molecules, atoms and the like. These data structures typically hold structural or experimental data that is read from disk, so we need support for *import and export in various file formats*. In many applications, this data is then manipulated using *Molecular Mechanics* methods. These are empirical models describing properties and dynamic behavior of matter at the atomic level. Finally, the *visualization* of the resulting structures is usually desirable.

### 6.2.3 Openness

Since we cannot provide all functionality and we also do not want to reimplement existing algorithms, it was also an important aim to provide a sufficient *openness*, meaning compatibility with other class libraries. For example, from the very start we planned the integration of LEDA [78], ABACUS [59], and CGAL [31].

Furthermore, openness includes *extensibility* and *modularity*, *i.e.* it should be simple to add new functionality and data structures without changing the existing code.

### 6.2.4 Robustness

The term robustness describes the code's ability to cope with unexpected or faulty data. A good example of such faulty data is the PDB file format [9] for crystallographic structures. This format is the most common format for structural data and is well standardized. Nevertheless, only very few files circulating among scientists adhere to this standard for historical as well as for practical reasons. Strange as it may seem, but the implementation of a robust code for reading these files is a very demanding task. Implementing a file reader accepting standard compliant files is simple enough, but is not sufficient as only very few files could be read. Reliable reading of non-compliant files requires quite some biochemical knowledge and non-trivial heuristics to extract as much of the data as possible.

According to the 80-20 rule [81], which states that a typical program spends 80 percent of its time executing 20 percent of the code, we decided to optimize only the computationally demanding code sections and sacrifice a bit of performance to an overall robust and reliable code. Since robustness also implies correctness, we also require techniques for software testing.

## 6.3 Choice of Programming Language

One of the first decisions was the choice of the programming language. As discussed in the last section, we need a widely used programming language. Most applications in Molecular Modeling are written in FORTRAN, mainly for historical reasons and because most chemists, physicists, and biologists only know this language.

Although FORTRAN, especially High-Performance FORTRAN (HPF), is very well suited for numerically demanding tasks and often outperforms code written in any other high-level language, we decided against FORTRAN. Among our design goals were extensibility and modularity. Both are very hard to achieve in a procedural language; object-oriented (OO) languages are clearly superior to achieve both goals, as for example Coulange [23] and Meyer [79] point out in detail. So we were basically left with the choice of Java or C++. The advantages of Java are the higher portability and the increased robustness (*e.g.* no pointers). C++ on the other hand was the better known and more widely spread language. C++ code also generally outperforms Java code, and C++ allows generic programming as well as operator overloading. Finally, at the time we made our decision, all class libraries we wanted to use along with BALL were implemented in C++.

With the ANSI C++ standard [5] still under development, only a small subset of its features was available to us while developing BALL. For example, the use of member templates and the integration of the standard template library (STL [83]) was introduced at a rather late stage of the development. Even with the final ANSI standard passed, the integration of these language features was delayed due to lacking standard compliance of most C++ compilers.

## 6.4 Architecture

The overall structure of BALL may be seen as consisting of several layers (Fig. 6.1). The lowest layer contains the so-called *Foundation Classes*. These classes implement a set of general data structures like an extended string class, hash maps, or design patterns. The implementation of this layer is based on the standard template library.

The second layer consists of the *kernel classes*. They implement the fundamental molecular data structures, *e.g.* atoms, molecules, bonds, *etc.* The kernel classes depend on the foundation classes and are required for the implementation of the third layer.

The third layer consists of several *basic components*. Each of these components provides functionality for a clearly defined field of Molecular Modeling and is not physically dependent on any of the other basic components. The basic components are implemented using the kernel classes and the foundation classes. Besides the key functionality we identified in Section 6.2.2, we also provide support for the search for structural similarities, for the calculation of nuclear magnetic resonance (NMR) spectra, and some solvation methods. The visualization component relies on OpenGL [86] for platform independent 3D graphics and on QT [102] for a portable graphical user interface (GUI).

BALL classes can also be used as extensions in an object-oriented scripting language (Python [121]) and support is also provided to embed this scripting language into BALL applications (see Section 6.8).

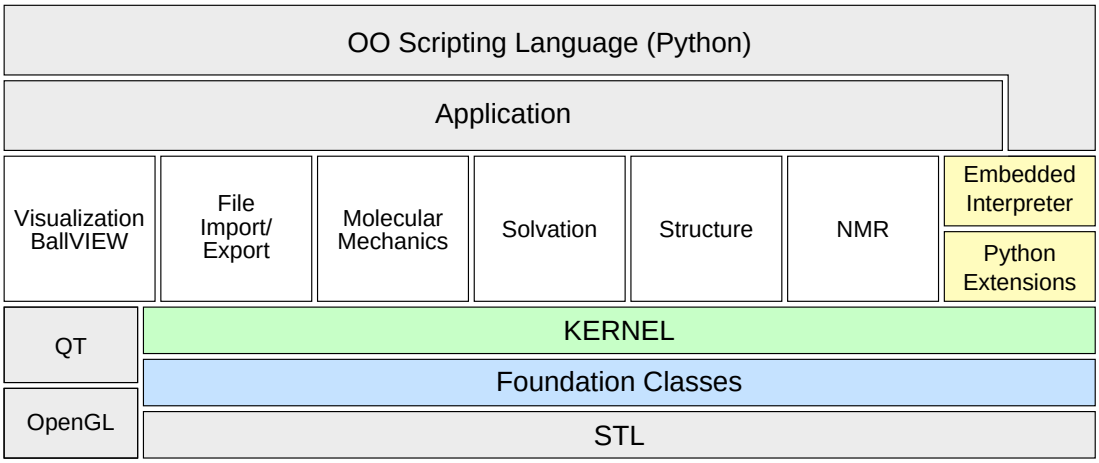



Figure 6.1: *Overview of the structure of BALL. The basic components are in white, the scripting language support in yellow and external libraries that are not part of BALL are in gray.*

The architecture and the high-level design of the three layers will be described in more detail in the next three sections. Due to the large number of classes in BALL, we had to restrict this overview to some of the more interesting examples.

## 6.5 The Foundation Classes

This section describes a selection of the most important foundation classes as well as some common definitions and functions required for the implementation of the kernel and the foundation classes.

### 6.5.1 Global Definitions

BALL defines a number of global type definitions and constant definitions. The types that are used in BALL are declared in `COMMON/global.h`. These definitions are typedefs to primitive types as determined by configure. For example, BALL uses the type `PointerSizeInt` to describe an integer type with the same bit width as the pointers on the respective platform. `configure` determines the correct type for this definition and includes it into `config.h` which is included by `COMMON/global.h`. By defining these types we achieve a better portability. Especially the compatibility between 32 bit systems and 64 bit systems relies heavily upon the use of these types (see also the implementation of object persistence in Section 6.5.3).

The namespace `Constants` holds a large number of constant definitions. These are mainly physical and chemical constants (*e.g.* speed of light, electron mass, etc.) and mathematical constants (*e.g.* $\pi$, e). The BALL kernel uses only these constants, thus reducing the possibility of errors due to wrong constants. This also reduces numerical inconsistencies between different code parts.

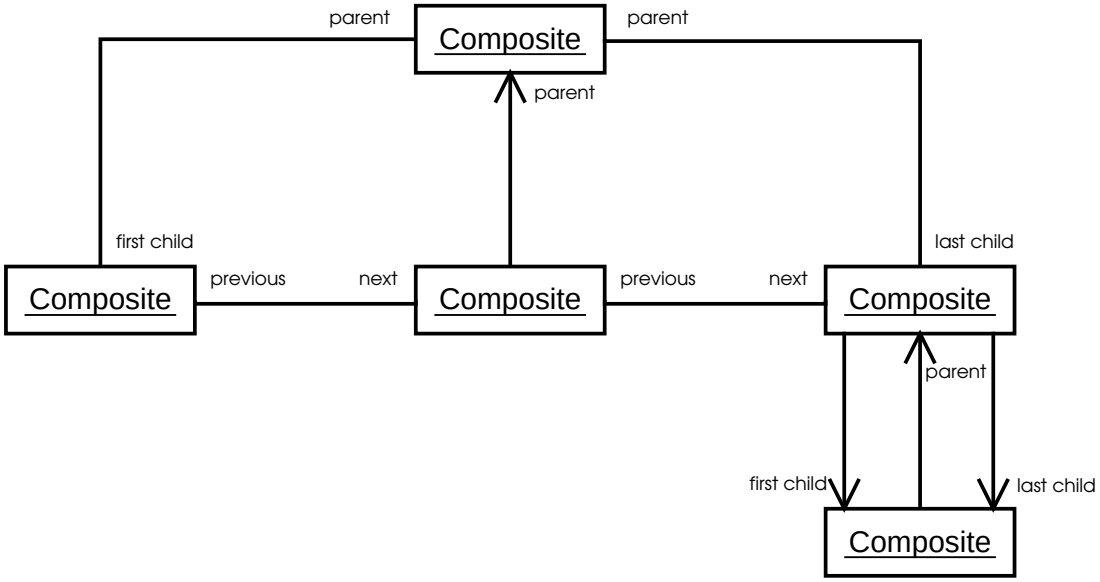



Figure 6.2: *The tree structure formed by multiple instances of* `Composite`

### 6.5.2 Composite Class

The `Composite` class implements the *composite design pattern* [41]. Each composite contains a pointer to its parent composite, a pointer to the first child, a pointer to the last child, a pointer to the previous composite, and a pointer to the next composite. Thus, the children of each composite form a doubly linked list that can be accessed from both ends (pointer to the first child and to the last child). The doubly linked list structure is required to iterate efficiently forward as well as backward. Both cases occur rather frequently in our applications. The structure of the resulting tree can be seen in Fig. 6.2.

The `Composite` class is essential to the kernel, because it represents the base class of most kernel classes (see Section 6.6). Due to the flexibility and functionality required from the kernel, the `composite` class implements a lot of functionality; in fact, it is one of the most complex foundation classes. It provides a large number of methods to manipulate tree structures (*e.g.* insertion and deletion of children or splicing of subtrees).

#### Selection

The `Composite` class is also derived from `Selectable`, so each node of a tree may be either selected or deselected. We use this selection mechanism throughout the whole kernel. Because one of the most frequent operations is a test whether any node in a subtree is selected, this operation had to be implemented very efficiently. The corresponding method `containsSelection` answers the question in constant time by simply returning the value of the attribute `contains_selection_` of `Composite`. By reimplementing the virtual methods `select` and `deselect` (inherited from `Selectable`), the selection or deselection is propagated up-

wards in the composite tree and all `contains_selection_` flags are consistently updated. Since the depth of our trees is small (usually below 5), this update operation can be done rather efficiently.

#### Time stamps

Since `Composite` is the base class for most kernel classes, we had to conceive a number of techniques to combine efficiency and robustness in the kernel. One of these techniques is the use of *time stamps*. For example, when performing an MD simulation or an energy minimization, we have to construct temporary internal data structures from the kernel data structure to increase the efficiency. These minimization or simulations are usually not performed in a single step, but the user might choose to inspect the system he was simulating after defined time intervals. If he decides to change either the selection or the topology of the system, the internal data structures of the simulation become inconsistent with the system and have to be rebuilt. There are basically two ways to solve this problem: First, we might choose a robust implementation that rebuilds its internal data structures each time it regains control of the system, which is very inefficient. The second strategy is a "don't care" approach. We assume the user did not change the data structures, and if did, he should be responsible for the consequences. This strategy clearly reduces the robustness of the code.

Using time stamps, we can achieve a robust and efficient implementation. Each `Composite` object contains two time stamps (class `TimeStamp`): the selection time stamp and the modification time stamp. A `TimeStamp` object simply stores a point in time. Using the method `stamp` this point in time can be updated to the current time. All methods that update the selection in a composite change the selection time stamp and all methods that change the topology of the composite tree update the modification time stamp. Similar to the selection flags, time stamps are propagated upwards in the composite tree. Thus, changing an atom in a system updates the time stamp of the system as well. The simulation classes now simply have to store the time of creation of all internal data structures in a time stamp. If this time stamp is older than the selection or the modification time stamp of the system, the internal data structures are rebuilt.

### 6.5.3 Object Persistence

Object persistence is an important concept in object oriented programming. It allows the storage of object instances beyond the life time of the applications that created these objects and an exchange of these objects between independent applications. The process of storing persistent objects is also called *serialization*. The format employed for object storage should be portable to allow the exchange of objects between different platforms. Serialization of objects should be easy to use and easy to implement for user defined objects as well.

#### Problems with serialization in C++

While several OO languages provide support for the serialization of objects (*e.g.* Java or Python), it is a tiresome subject in C++. There are mainly two problems related to this subject: multiple inheritance and static class attributes. Fig. 6.3 shows the so-called *deadly diamond*

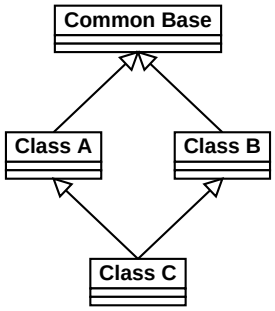



Figure 6.3: *Multiple inheritance: the "deadly diamond"*

inheritance. In this example, class `C` is derived from `A` as well as from `B`. `A` and `B` are both specializations of a common base class `CommonBase`. This causes `C` to contain two instances of `CommonBase`: one inherited from `A` and one inherited from `B`. This causes fundamental confusion in class `C`, since it is unknown which of these instances of `CommonBase` is meant when accessing methods or attributes. In C++, this ambiguity can be resolved using *virtual inheritance*. Nevertheless, virtual inheritance leads to another problem: it is not possible to cast down from the virtual base class to the derived class.

To avoid these problems, each persistent BALL class is required to be derived from a common base class (`PersistentObject`) exactly once. Multiple or virtual inheritance from this base class is not allowed.

The second problem are static class attributes. For example, the class `Object` defines a unique handle for each object (application wide). `Object` contains a member `handle_` and a static member `global_handle_`. `global_handle_` is incremented each time a constructor of `Object` is called and its current value is stored in the object's `handle_`. These handles serve as a unique identification for an object instance. When storing these objects, it is obviously not desirable to store `global_handle_`, because its retrieval would reset the global handle counter and could result in duplicate handles. There is no general solution for the problems caused by static object attributes, the best rule is to avoid them wherever possible and decide on a case-by-case basis where it is unavoidable.

### Detailed design

We will now sketch the mechanisms provided for persistent object storage in BALL. An overview of the participating classes and their relationships is given in Fig. 6.4.

The base class for all persistent objects is `PersistentObject`. It provides three virtual methods: `persistentRead`, `persistentWrite`, and `finalize`.

Storing and retrieving a persistent object is controlled by the class `PersistenceManager`. This class constructs a persistent representation of an object and writes it to an `ostream` or may, vice versa, retrieve a serialized object from a stream. Hence, `PersistenceManager` implements the *builder* design pattern and the *factory method* design pattern [41].

The builder pattern is intended to separate the construction of a complex object from its

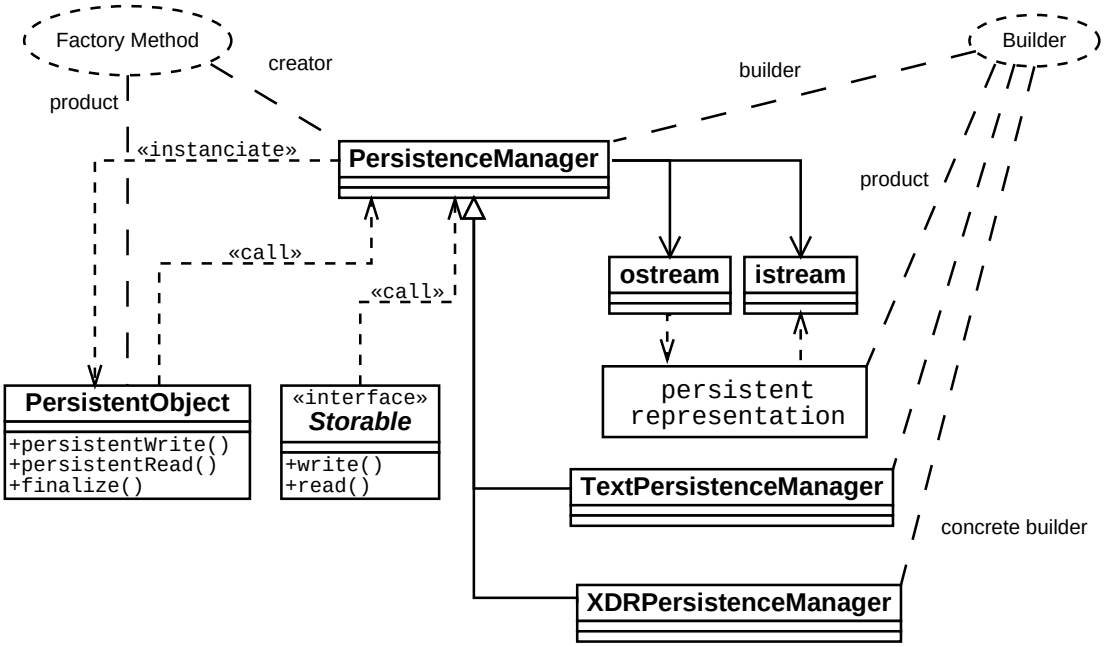



Figure 6.4: *Object persistence in BALL: an overview*

representation. Thus, the same construction process can create different representations for the object. The builder pattern describes the persistent storage functionality of `PersistenceManager` rather accurately. The `PersistenceManager` (the *builder* in terms of the design pattern) defines an interface for the construction of a persistent and portable representation of objects. There are two derived classes available (`TextPersistenceManager` and `XDRPersistenceManager`) that implement a *concrete builder*, *i.e.* an object that constructs a representation of the persistent object. The persistent representation is called the *product*. The fourth participant of the design pattern, the *director* is represented by the application.

The second pattern involved is the factory method pattern (also known as *virtual constructor*). This pattern defines an interface for object creation. The decision which class to instantiate is deferred to subclasses. The participants in this pattern are a *product*, a *concrete product*, a *creator*, and a *concrete creator*. The product defines the interface for the instantiated objects (here: `PersistentObject`), while the concrete product represents the constructed class itself (here: any class derived from `PersistentObject`). Similarly, the creator declares the factory method returning an object of type product, while the concrete creator implements this factory method. The creator is represented by `PersistenceManager` and the concrete creator is represented by any of its derived classes.

As we use multiple inheritance rather frequently in the kernel, it was difficult to avoid multiple inheritance from `PersistentObject`. For example, the `Composite` class is derived from `Selectable`. This class basically provides an additional boolean flag and some accessors to select and deselect an object. For the reasons stated above, it is not possible to derive `Selectable` from `PersistentObject` as well. However, it should be possible to store

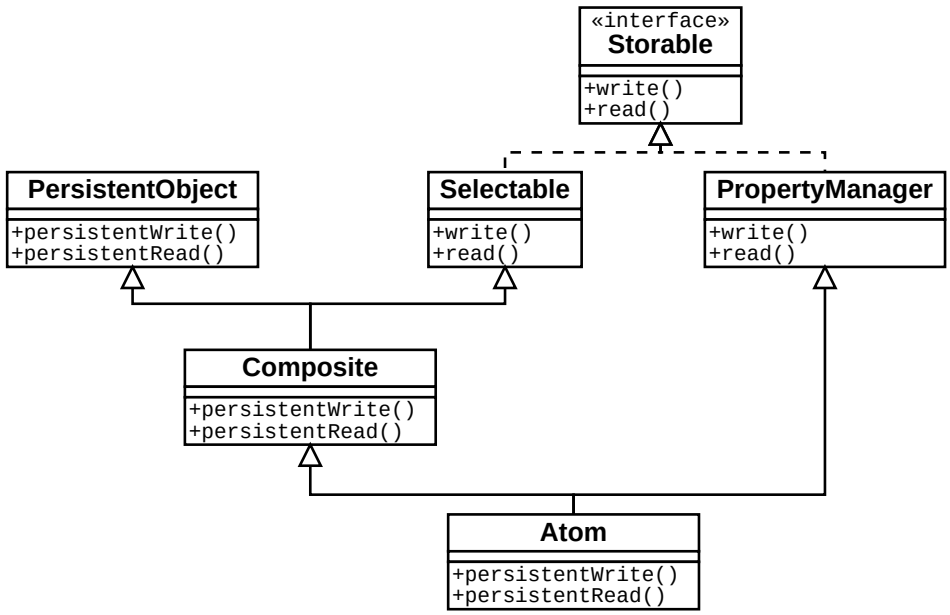



Figure 6.5: *The `Atom` class and its base classes.*

the contents of an instance of `Selectable` in a convenient way. This can be achieved by defining an interface that contains the two methods `read` and `write`. These two methods can be used in the implementation of `persistentRead` and `persistentWrite` for storing and retrieving the contents of the base class. Furthermore, it is possible to derive from several `Storable` classes without running into problems due to multiple inheritance. The cooperation between `PersistentObject` and the `Storable` interface is illustrated in the class graph of the `Atom` class (Fig. 6.5).

### Implementation

`TextPersistenceManager` provides three different layers of methods. Layer 0 provides fundamental methods to store and retrieve primitive data types (*e.g.* int, float, pointer, string). For each primitive type, there is a `put` method and a `get` method. The layer 0 methods implement the format of the persistent representation that is produced. They are implemented by subclasses of `PersistenceManager`. These methods determine the format of the persistent representation (*e.g.* text or binary).

The layer 1 commands are needed to implement the `persistentWrite` and `persistentRead` methods. They are implemented in `PersistenceManager` using the layer 0 commands. Knowledge of these methods is only required to implement new persistent classes.

To store and retrieve existing objects, the user has to know the level 2 commands alone. They provide methods to assign the input and output streams of the persistence manager and to store and retrieve objects.

### Portability

BALL has been ported to a variety of different platforms. The main differences between these platforms are

- word width (32 bit vs. 64 bit machines)
- byte order (big endian vs. little endian)
- floating point formats

A portable implementation of object persistence has to find platform independent representations for all data types to overcome these differences. First, all pointers are represented internally (in the `PersistenceManager` class) using 64 bit types and are also stored this way. Byte order and floating point formats affect the portable storage of primitive data types. Therefore, the implementation of the layer 0 commands in any concrete persistence manager has to ensure portability. One way to achieve this is the use of text-based formats. This approach is implemented in `TextPersistenceManager`. This class stores an object in a human readable text format. It uses ASCII representations of numbers to achieve portability. However, the resulting representation is large and slow to parse. This class is mainly intended for debugging.

The class `XDRPersistenceManager` produces a much more compact binary format. It is based upon the *XDR* format (*External Data Representation*). XDR is a standard for portable data exchange, introduced by SUN Microsystems [115, 111]. It is available on all platforms and was introduced to allow data exchange via the network. For example, remote procedure calls (RPC) use XDR as their data exchange format. XDR provides methods for packing primitive data types (*e.g.* int, float, string) into four byte packets. These packets are then written to a stream. The resulting representation is not human readable, but has little overhead and can be processed rapidly. `XDRPersistenceManager` translates the layer 0 commands of `TextPersistenceManager` to the corresponding XDR function calls and converts the XDR buffers to a C++ stream and back.

### User interface

Using persistence in BALL is rather simple. It requires the instantiation of a persistence manager, assignment of an input or output stream, and finally the persistent reading or writing of the object.

```
1 | Protein p = ....;                              // the object to serialize
2 | ofstream outstream("test.pers", ios::out);     // open output stream
3 | TextPersistenceManager pm(outstream);          // create persistence manager
4 | pm << p;                                       // serialization
5 | outstream.close();                             // close the output stream. Done.
```

Deserializing a persistent object is equally simple:

```
1 | ifstream instream("test.pers");                // open input stream
2 | TextPersistenceManager pm(instream);          // create persistence manager
3 | PersistentObject* po;                         // define a pointer to a PO
```

```
4   pm >> po;                                    // deserialization
5   instream.close();                            // close the input stream
6
7   if (po != 0)                                 // identify the object
8   {
9     cout << "Read␣object␣of␣type␣" << typeid(*po).name() << endl;
10  }
```

Instead of using file streams (as in the two examples above), it is possible to use any fully transparent stream derived from `istream` or `ostream`. For example, BALL uses the `SocketStream` class to exchange persistent objects between different applications using TCP sockets. This feature is also used by `molview` to exchange objects between a client application and the visualization server (see Section 6.7.4).

### 6.5.4 Run-Time Type Identification

ANSI C++ provides two operators for the Run-Time Type Identification: `dynamic_cast` and `typeid`. With the aid of these two operators it is possible to identify the type of any object at run time. A typical application of `dynamic_cast` may look like this:

```
1   Molecule* molecule = ....;
2   if (dynamic_cast<Protein*>(molecule) != 0)
3   {
4     // cast to protein and perform some protein specific operations
5   }
```

Here, dynamic cast is used as a predicate. It identifies whether a pointer to a given instance of `Molecule` is in fact an instance of class `Protein`, a derived class. If this is true, some code, which is only applicable to proteins, is executed. Similar code fragments are encountered throughout the BALL kernel. However, the function of the code is not obvious at the first look. Therefore, we decided to implement a set of template wrapper functions to simplify the Run-Time Type Identification. They allow us to write code with a better readability, like

```
1   Molecule molecule = ....;
2   if (isKindOf<Protein>(molecule))
3   {
4     // cast to protein and perform some protein specific operations
5   }
```

Theses functions have been collected in a common namespace (`RTTI`). The most commonly used functions are `isKindOf<T>` and `isInstanceOf<T>`. `isKindOf<T>` is parameterized with a class `T` and returns **true** for instances of `T` and instances of classes derived from `T`. Similarly, `isInstanceOf<T>` returns **true** for instances of `T`, but **false** for instances of classes derived from `T`.

Other functions in the `RTTI` namespace provide unique IDs for each class, a unique name, and static default instances of arbitrary objects.

### 6.5.5 Iterators

Several of the BALL classes are container classes, *i.e.* they contain multiple instances of objects. It is often necessary to iterate over all elements of a container object. In the Standard

Template Library, this is done via *iterators*. We use the same approach and a syntax similar to that of the STL for the iterators in BALL. The implementation of iterators is simplified by a common template base class for all iterators and the use of the *iterator traits* technique [85]. We give examples of the use of iterators in the BALL kernel in Section 6.6.2.

### 6.5.6 Processors

A recurring problem in OO design is the separation of algorithms and data structures, especially in an extensible framework like BALL. The need for functionality often entails the danger of *interface pollution* (see *e.g.* [73] for a discussion), *i.e.* fat interfaces that are cluttered with methods required by a number of client classes, but not necessaryly by the class itself. Also the extensibility of the framework is seriously hampered if adding a new algorithm requires changes in the interface of the kernel classes. A classical solution to these problems is, for example, the *visitor pattern* [41]. This pattern is designed to add new operations without changing the interface of the classes it operates on.

In the BALL kernel data structures, the problem becomes even more troublesome: the kernel data structures are container that may contain polymorphous objects (*e.g.* atoms, residues, and molecules). Most of the algorithms in BALL that operate on kernel data structures should be applicable to all kernel classes (*e.g.* an algorithm should operate on molecules as well as on a single atom). Furthermore, most of the algorithms in BALL require an iteration over all elements (*e.g.* atoms) of a kernel object. Hence, we combined the advantages of the visitor pattern and the iteration capabilities of the kernel to a common concept: the *processor* concept.

Processors are objects that implement an algorithm and can be *applied* to any kernel data structure. All processors are derived from the template class `UnaryProcessor<T>`. They are parameterized with the type they operate on. For example, a processor to calculate the geometric center of any kernel object just has to know the atom coordinates, so the processor is derived from `UnaryProcessor<Atom>`. The kernel data structures implement an interface to arbitrary processors through the template member function `apply<T>`. Thus, arbitrary processors can be *applied* to arbitrary kernel classes. Implementing a processor is a rather trivial task. Generally, a processor provides three methods: `start`, `finish`, and `operator ()`.

The `apply<T>` method of a kernel container class first calls `start`, then it iterates over all objects of type `T` it contains. For each of these objects, it calls `operator ()` of the processor. Thus, the processor "sees" just the objects it is interested in. At the end of the iterations, `finish` is called to inform the processor that he cannot expect further objects. While `start` usually performs some initialization steps, `finish` can be used for final actions (freeing allocated memory, calculating results).

We will now illustrate the processor concept by means of a simple example: the `CenterOfMassProcessor` which calculates the center of mass for any given kernel object. The center of mass $\vec{R}$ of $N$ particles (atoms) of mass $m_i$ and position $\vec{r_i}$ is defined as follows: [43]

$$\vec{R} = \frac{1}{M} \sum_{i}^{N} m_i \vec{r_i} \quad \text{with} M = \sum_{i}^{N} m_i \tag{6.1}$$

So we have to calculate the sum over all atom masses and the products of mass and position for all atoms. Since we operate on atoms only, we derive `CenterOfMassProcessor` from `UnaryProcessor<Atom>`. The start method is trivial. It just initializes the member variables `mass` and `center` to zero:

```
1  bool CenterOfMassProcessor::start()
2  {
3    mass = 0.0;
4    center = Vector3(0.0, 0.0, 0.0);
5
6    return true;
7  }
```

The `operator ()` does the true work: for each atom it is called with, it determines the mass and the position and calculates the two sums:

```
1  Processor::Result CenterOfMassProcessor::operator () (Atom& atom)
2  {
3    float atom_mass = atom.getElement().getAtomicWeight();
4    center += atom.getPosition() * atom_mass;
5    mass += atom_mass;
6
7    return Processor::CONTINUE;
8  }
```

Finally, we implement `finish` as follows:

```
1  bool CenterOfMassProcessor::finish()
2  {
3    center /= mass;
4
5    return true;
6  }
```

We may now apply our processor to calculate the center of mass of a molecule `m`:

```
1  Molecule m = ...;
2  ...
3  // create an instance of the processor
4  CenterOfMassProcessor center_proc;
5
6  // apply it to the molecule
7  m.apply(center_proc);
8
9  // print the center of mass
10 cout << "Center␣of␣mass␣is␣at␣" << center_proc.center << endl;
```

### 6.5.7 Options

Algorithms in Computational Molecular Biology tend to require huge numbers (sometimes several dozens) of arguments and parameters. This usually leads to bulky interfaces and reduces the modularity, because related methods have different interfaces and cannot be exchanged easily. A solution to this problem is the `Options` class. Instead of passing dozens of parameters or polluting the class interface with dozens of member functions, the class contains an `Options` object. Each `Options` object may hold an arbitrary number of key/value pairs.

An example from the solvation component (the Finite Difference Poisson-Boltzmann class – FDPB) will illustrate its use. The FDPB class performs a number of non-trivial calculations. There are currently 22 different parameters used to adopt the behavior of the class to specific problems. This would require 44 member functions to set or get the parameters (if data encapsulation is taken seriously). Using the Options class, FDPB gets just one more member called options and one struct containing symbolic names for the options. Setting one of these parameters (*e.g.* the grid spacing) looks as follows:

```
1 | FDPB fdpb = ...;
2 | fdpb.options[FDPB::Option::SPACING] = 0.5;
```

Again, operator overloading provides a clean and comprehensible interface.

Another advantage of the Options class is their ability to store options to a file and retrieve them from a file. This has proven especially useful to document the parameters used in a specific experiment during large-scale parameter optimizations.

### 6.5.8 Logging Facility

Printing error messages, warnings, or progress information are very simple ways to communicate with the user. Problems arise if this information has to be stored, if the application has a graphical user interface, or if the application runs distributed on a network. In each of these cases a standardized logging facility is required to allow redirection of the output to the users' terminal or a log file.

The class that implements this logging facility is called LogStream. This class is derived from ostream, so it can be used in the same way as the C++ streams cout or cerr. There is one global instance of LogStream which is globally defined and is used throughout the BALL kernel to log error messages. Each message is stored along with a time stamp and an error level (ranging from information to error). The user can also attach arbitrary stream objects (*e.g.* files) to store the information or can retrieve stores error messages. It is even possible to trigger certain actions as soon as messages with a specific error level occur. In applications like molview (the stand-alone visualization tool) the LogStream class allows the convenient redirection and central administration of debugging, error, or progress messages.

The following code section gives an impression of the usage of the LogStream class:

```
 1 | // initialize the global LogStream
 2 | Log.setPrefix(cout, "[%T] ");
 3 | Log.setPrefix(cerr, "[%T] ERROR: ");
 4 |
 5 | ....
 6 | //  print some informational message
 7 | // (level INFORMATION is associated with cout by default)
 8 | Log.info() << "starting calculation..." << endl;
 9 |
10 | ....
11 |
12 | // print an error message
13 | // (level ERROR is associated with cout by default)
14 | Log.level(LogStream::ERROR) << "error -- execution aborted!" << endl;
```

The two streams cout and cerr are associated with the global instance of LogStream,

`Log`, by default and catch all messages with information level (cout) and all error messages (cerr). In lines two and three, we assign a prefix to each of the streams via the `setPrefix` method. The string `"%T"` is expanded to the current time, similar expressions exist for the date, the error level, the line number, and so on. The global log stream can be used as any C++ stream, as can be seen in lines 8 and 14. In line 8 we print a message with informational level (via the method `info`, which returns an object of type `ostream`). The resulting message is hence prefixed by the current time and redirected to `cout`. The error message printed in line 14 is prefixed by the time and the string error (as specified in line 3) and redirected to `cerr`, the stream associated with error messages. The output of the above code thus looks like this:

```
1   [11:19:02] starting calculation...
2   [11:19:05] ERROR: error -- execution aborted!
```

### 6.5.9 Strings and Related Classes

Since many applications require extensive string operations (often to read or transform text-based input files), BALL provides a powerful `String` class and regular expressions. The string class is derived from the STL string class to give an identical interface and to take advantage of the efficient and flexible implementation. We added a lot of functionality to simplify the usage of the string class. The `String` class enables the user to access whitespace-separated fields of the string (similar to the definition used by the unix program `awk`) and simplifies the parsing of field based file formats.

The `String` class is complemented by the `Substring` class that allows an efficient manipulation of substrings and the `RegularExpression` class that allows matching and manipulation of strings through regular expressions.

### 6.5.10 Mathematics

BALL provides a number of classes and functions related to mathematics. The classes fall into two categories: fundamental data structures and geometric objects. The first category includes simple vectors, matrices, quaternions, angles, *etc.* The geometric objects are used to represent three-dimensional geometric primitives (*e.g.* boxes, spheres, *etc.*) and serve as a basis for the implementation of the visualization component. All these classes are parameterized with a number type (*i.e.* they are template classes). The generic implementation allows us to use different numerical types for different problems. For most applications, single precision floating point numbers are completely sufficient, but we can also use LEDA big floats if that precision is required.

### 6.5.11 Miscellaneous

It would be tiresome and not very instructive to enumerate all remaining foundation classes, consequently we will just mention some of the more important categories of classes. First, there are exceptions. The BALL error handling is based on a class hierarchy of exceptions that are all derived from `GeneralException`. BALL also implements its own handling of uncaught

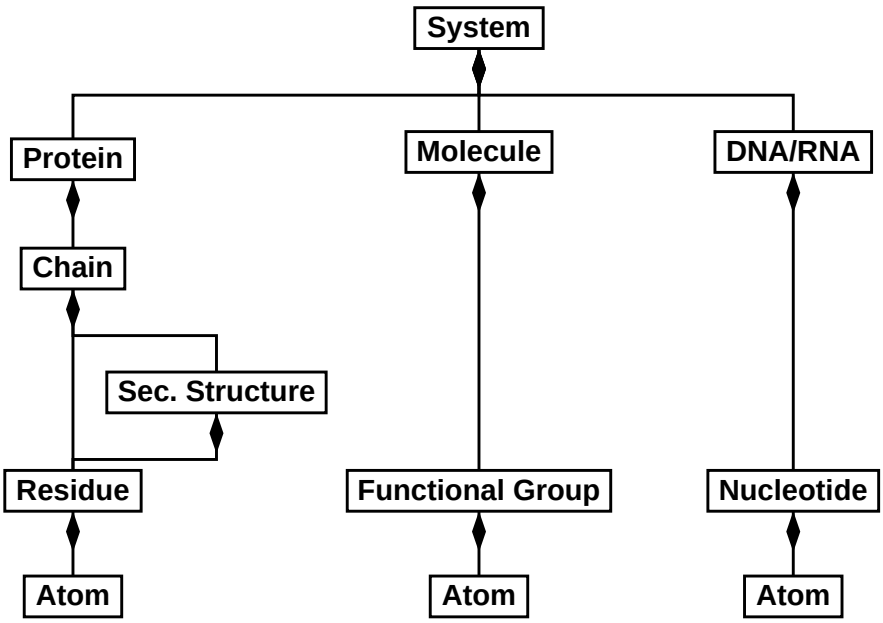



Figure 6.6: *Hierarchical structures in biochemistry: the problem domain*

exceptions to give the user as much information about the error as possible. Therefore, all exceptions document the line and the file where they are thrown and provide a short error description. This information is printed whenever an uncaught exception occurs.

Additionally, BALL provides a whole range of generic basic data structures. There are several flavors of trees, hash associative containers, or grid data structures. Then, there are classes related to the operating system. They are used to implement a portable, abstract interface to files, directories, system time, or network sockets.

## 6.6 The Kernel

### 6.6.1 Molecular Data Structures

The second layer is the so-called kernel and implements the molecular data structures. Since one of our design goals was intuitive usability, we spent a lot of effort on the design of these classes. Our first step was an analysis of the objects we wanted to describe. We found that the biochemical domain is well described by a hierarchical model. All the objects we want to represent consist of *atoms*. In chemistry, several atoms often form *functional groups* from which *molecules* are built. A set of molecules is then called a *system*. This hierarchy of entities is what we call the molecular domain (Fig. 6.6).

One of our main concerns was an intuitive usability of the kernel class hierarchy especially for users with solid biochemical knowledge. Therefore, our kernel classes should directly reflect the hierarchical relationships found in biochemistry. The molecular domain is quite simple. Obviously, there is a strong hierarchical ”has-a” relationship between these entities.

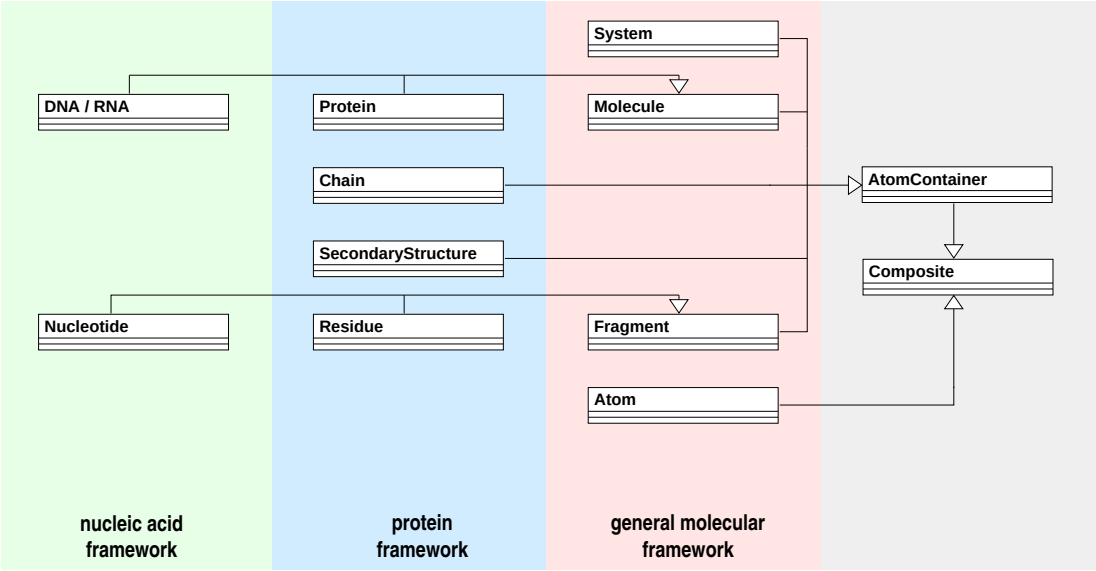



Figure 6.7: *The kernel class hierarchy*

This hierarchical structure is best described by a tree structure. A system may thus contain molecules, consisting of functional groups containing atoms. The class names are chosen accordingly as `System`, `Molecule`, `Fragment`, and `Atom`. We chose the term fragment instead of functional group as this term is more general and more accurate when transferred to the protein and nucleic acids domain. Similar relationships also exist in the protein domain and the nucleic acid domain: a protein contains several chains, each of these chains may consist of secondary structures built from residues consisting of atoms. In the nucleic acid domain everything is quite neat and simple: there are two kinds of nucleic acids – DNA and RNA – each consisting of nucleotides.

In Fig. 6.6 one can also identify a ”horizontal” relationship between the entities of the different domains: a protein is by definition also a molecule, as is a nucleic acid. Similarly, residues and nucleotides can be seen as fragments of proteins or nucleic acids. These fragments in the biopolymers correspond to the functional groups in a general molecule – thus the class name `Fragment` instead of `FunctionalGroup`. This ”is-kind-of” relationship can be modeled by inheritance.

The ”has-a” relationships, which led to the tree structure, may be modeled using the *composite* design pattern [41]. Its intent is described as follows:

> “Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.”
> *[41], p. 163*

The pattern is thus well suited to represent the hierarchical tree structures required for the kernel and nevertheless defines a uniform access to all classes. This uniform access is required for most algorithms, since they should be able to operate on molecules as well as on any other kernel data structure.

The molecular domain thus results in the *molecular framework* (see Fig. 6.7). The entities of the protein domain are represented by the classes `Protein`, `Chain`, `SecondaryStructure`, and `Residue` in the protein framework. Finally, the nucleic acid framework consists of the classes `NucleicAcid`, `DNA`, `RNA`, and `Nucleotide`.

All these classes have a common base class: `Composite` which implements the composite design pattern (see Section 6.5.2). All fragment-like classes are derived from a common base class (`AtomContainer`) that implements most methods that are common to classes containing atoms.

### 6.6.2 Iterators

Iteration over kernel data structures should be as simple as iterating over any STL container type. However, there is a fundamental difference: the BALL kernel classes are not simple containers, but they can contain a variety of different objects. For example, a user should be able to iterate over all molecules of a system as well as over all atoms of the system. Since in C++ overloaded methods cannot be distinguished from their return type alone, the use of the typical STL `end()` and `begin()` methods is not possible. Instead, we had to implement specialized methods for each class. Similarly, we cannot simply define *one* iterator type for a kernel class, but we have a set of iterator classes that are defined independently. Iterating over all atoms of any kernel data structure requires an iterator of type `AtomIterator`. An iterator pointing to the first atom of a `System` is returned by the method `beginAtom`, a past-the-end iterator is returned by `endAtom`. Apart from these restrictions, the use of kernel iterators is very similar to that of STL iterators:

```
1  System S = ...  // define a system
2
3  AtomIterator ai;
4  for (ai = S.beginAtom(); ai != S.endAtom(); ai++)
5  {
6    // print the position of the atom
7    cout << ai->getPosition() << endl;
8  }
```

The kernel classes define forward and reverse iterators and *const* variants of these iterators.

### 6.6.3 Selection

A recurring task in many applications is the definition of subsets of molecular data structures. For example, it should be possible to identify all atoms belonging to a certain chain, all amino acids of a certain type, or all side chain atoms of a protein. Problems of this kind are solved via the selection mechanism of kernel objects. As all kernel classes are subclasses of `Composite`, they inherit the selection capabilities of this class. BALL now provides a simple language to specify predicates for the selection of objects. The following example shall illustrate the usage of this language:

```
1  // a system containing a protein
2  System S = ....
3
```

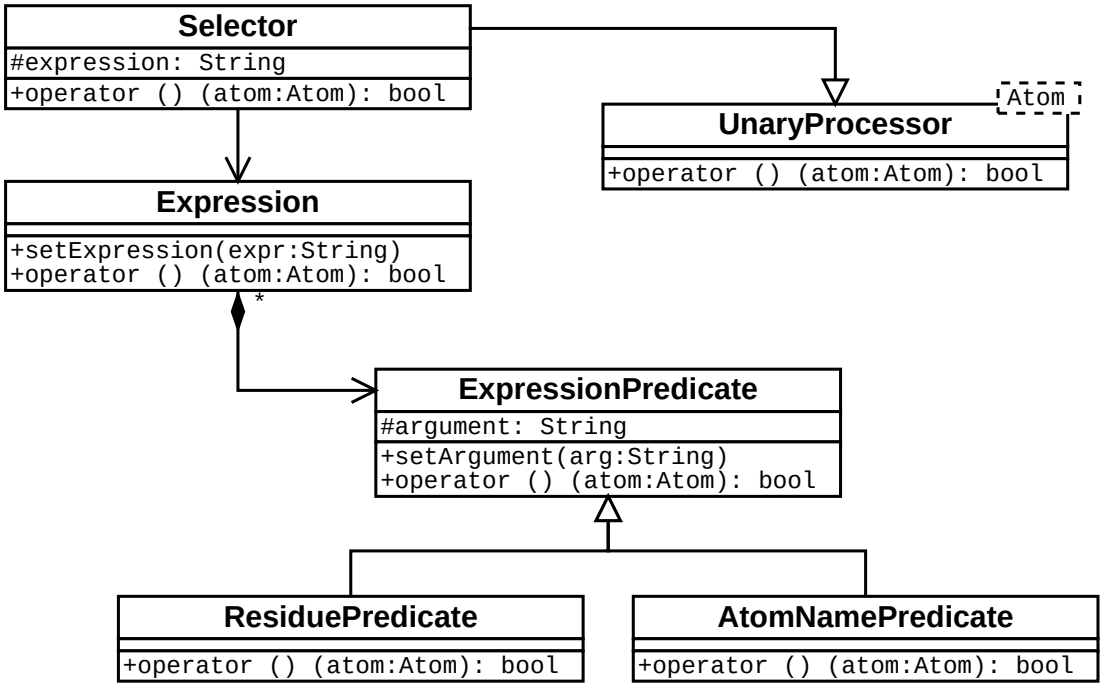



Figure 6.8: *The classes involved in the kernel selection mechanism.*

```
4 |  // create a selector that selects the Calpha atoms of
5 |  // all arginine (ARG) residues
6 |  Selector ARG_CA_select("residue(ARG) AND name(CA)");
7 |
8 |  S.apply(ARG_CA_select);
```

In Line 6, we create an instance of `Selector`. This class is a subclass of `UnaryProcessor<Atom>`, so we can apply it to arbitrary kernel objects. It is initialized with a string describing the desired selection. In our case, the selection of the $C_\alpha$ atoms in all arginine residues is required. Hence, we initialize the selector with a string describing this property. This string may consist of an arbitrary number of *predicates*, brackets, and the operators 'AND' and 'OR'. To determine all $C_\alpha$ atoms of all arginines, we have to combine the predicates `'residue(ARG)'` (which is true for all atoms inside residues of name 'ARG') and `'name(CA)'` (which is true for all $C_\alpha$ atoms) using the 'AND' operator. Each of these predicates corresponds to a class derived from `ExpressionPredicate` (see Fig. 6.8). An instance of `ExpressionPredicate` and its subclasses can decide whether the predicate they represent is true or false for a given atom. Each instance of `Selector` contains an `Expression` object that translates the string to a syntax tree containing the corresponding instances of subclasses of `ExpressionPredicate`. BALL provides a comprehensive set of predicates, the so-called *standard predicates*. These classes are known to the `Expression` class by default. The user may also create new predicates and register them with instances of the `Expression`. The implementation of new predicates is very trivial, because it just requires the implementation of a modified `operator ()`.

## 6.7 The Basic Components

A full description of the design of the basic components is beyond the scope of this work. Therefore, we will again describe only the most important and interesting classes of each basic component and give just a short overview of the remaining classes.

### 6.7.1 File Import/Export

When using experimental data (*e.g.* protein structures), the need for a standardized file format for data deposition and exchange is obvious. Unfortunately, the number of file formats used in Molecular Modeling alone is annoyingly high: the software package BABEL [124] currently reads 41 different molecule file formats! Not all of these formats are equally important, so we decided to implement only the most important ones. The information content of the file format varies widely. For example, the PDB format contains literature references, crystallographic data, and experimental conditions, while the XYZ format contains only coordinates. But all of these formats contain data on molecular structures. Therefore, we defined a uniform interface to all these file formats to be able to extract the structural data independent of the file format. This interface is derived from a common base class `MolecularStructureFile`. This interface is very simple to use and defines a minimal set of file operations: reading and writing of structural data from a `System`.

The subclasses of `MolecularStructureFile` inherit this interface and implement the reading and writing of structural data along with some file-format-specific data. We implemented classes for PDB files (`PDBFile`), HyperChem HIN files (`HINFile`), Sybyl MOL2 files (`MOL2File`), and XYZ files (`XYZFile`).

Through the use of operator overloading the interface became quite simple. Reading a PDB file can be done in just two lines of code:

```
1  ...
2  String filename = ....;
3  System S;
4  ...
5  // create a PDB file object and open for reading
6  PDBFile infile(filename);
7
8  // read the contents of the file
9  infile >> S;
```

Besides the molecular structure file formats, there is a variety of other file formats supported to store data of different kinds. For example, we use an XML-based format to represent hierarchical data (`ResourceFile`) or Windows INI files to store application settings and table-like data.

### 6.7.2 Molecular Mechanics

The fundamentals of Molecular Mechanics force fields have been described in detail in Section 1.5.2. Our prime interest in force fields is their capability to predict energies of proteins. Numerous different force fields have been developed over the years; we decided to implement

two of the more important representatives: the AMBER force field [22] and the CHARMM force field [15].

Both force fields were originally implemented in FORTRAN. The resulting applications are monolithic and difficult to maintain. We designed a modular approach to the implementation of force fields. The input of a force field is usually a set of atoms and bonds (a system) and a set of parameters. The force field defines a set of equations describing the energy of the system as a function of the atom coordinates.

An implementation of a force field has to provide two main parts: a *force generator* and an *energy generator*. The force generator calculates for each atom the forces caused by its interactions, while the energy generator calculates the total energy. Often overlooked – but in fact one of the hardest parts in the implementation of the force field – is the assignment of the force field parameters. For example, the AMBER force field defines about 900 different parameters. These parameters have to be assigned according to the atom types involved in a certain interaction. Hence, we also had to design a way to store these parameters and mechanisms to assign them automatically.

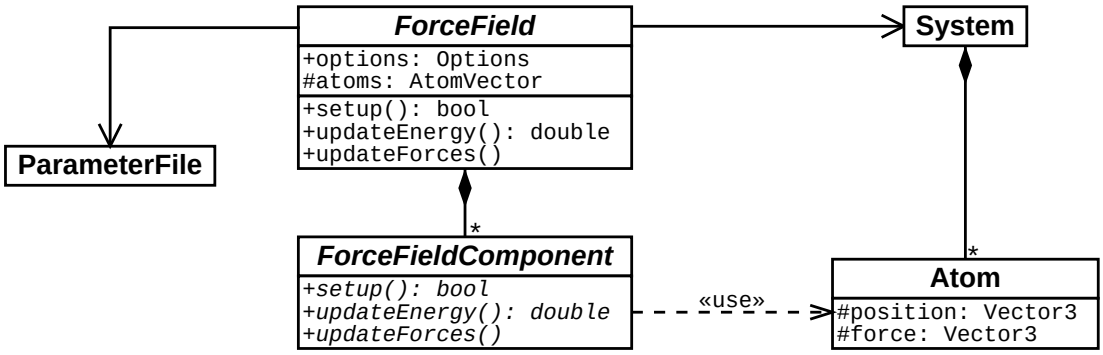



Figure 6.9: *The* `ForceField` *class and related classes*

The overall structure of the force field-related classes can be seen in Fig. 6.9. The most important class is the `ForceField`. It defines an interface to the force field without performing any calculations itself. Instead, it contains a list of `ForceFieldComponent`s. This class defines the interface for the calculation of a single interaction type (*e.g.* stretches or torsions). `ForceField` contains some attributes (and the corresponding accessors) that are needed for every force field. The attribute `options` holds options that are globally applicable to the whole force field. It also has a pointer to the system which contains the atoms. To achieve a better performance, these atoms are extracted and pointers to the atoms are stored in an `AtomVector` object.

The three most important methods defined in `ForceField` are `setup`, `updateEnergy`, and `updateForces`. The `setup` method performs some basic initializations for the force field, like extracting the atoms from the system, reading the parameters from a file, assigning these parameters, and setting up the force field components (by calling the `setup` method for each component).

The interface to the energy generator is defined by `updateEnergy`. This method calculates the total energy of the system as the sum of the energies contributed by each force field component. The component contributions are calculated by calling the `updateEnergy` method of each force field component. Similarly, `updateForces` calculates the forces by adding up the forces from each component.

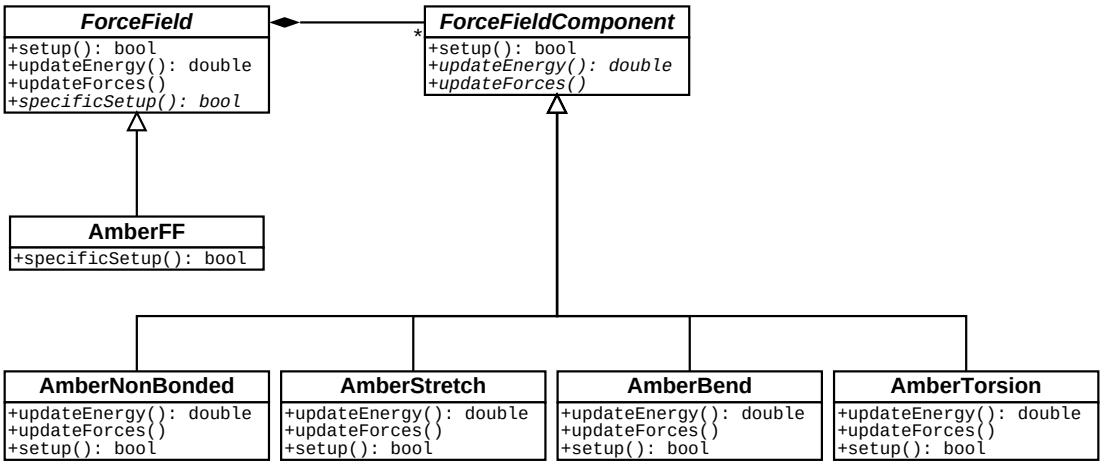



Figure 6.10: *The different energetic contributions (bond stretches, angle bends, torsions, and non-bonded interactions) are implemented in separate classes derived from* `ForceFieldComponent`.

A concrete forcefield is obtained by deriving a class from `ForceField` and one concrete force field component for each type of interaction. Fig. 6.10 shows the classes that implement the AMBER force field. The class `AmberFF` is responsible for instantiating the required force field components. In the case of the AMBER force field, there are four classes derived from `ForceFieldComponent`: `AmberNonBonded` (which implements the nonbonded interactions), `AmberStretch` (the bond stretch contribution), `AmberBend` (the angle bend contribution), and `AmberTorsion` (the torsion angle contribution). An instance of each of these classes in created by `AmberFF`. When `setup`, `updateEnergy`, or `updateForces` of `AmberFF` is called, the corresponding method is also called in each component. The implementation of a concrete force field is thus rather simple. Keeping the energetic contributions in separate classes leads to a better maintainability and reusability.

In contrast to our implementation, a number of existing force field implementations do not separate the force generator and the energy generator. The calculation of the forces and the energy is then performed in a single loop. This increases the performance for applications that require the simultaneous evaluation of both energy and forces (*e.g.* molecular dynamics simulations). We decided against this approach for two reasons. First, a separation leads to a better comprehensibility and maintainability of the code. Second, many of our applications (*e.g.* protein docking) only require the energetic evaluation, so the time spent on the force calculation can be saved.

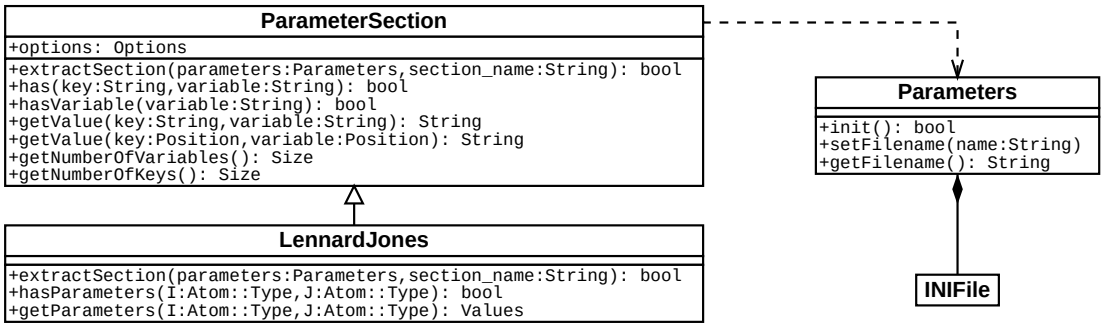



Figure 6.11: *The classes related to the reading of parameter files.* `Parameters` *contains an instance of* `INIFile` *which reads the parameter file. An instance of* `ParameterSection` *(or one of its derived classes) interprets the contents of one of the sections of the file as table-like data. The derived classes of* `ParameterSection` *are able to interpret this data and offer a comfortable interface to access it.*

### Parameter Assignment

A very tiresome and difficult process is the correct parameter assignment and the conversion of existing parameter files. The original implementors of the force fields usually provide a multitude of different versions of parameter sets that evolved over time. The formats used to store the parameters are text based and column oriented (FORTRAN style). We decided to use a common file format for all parameter files. This allows us to implement the input/output routines for these files independent from the force field. Furthermore, our file format has a much better readability, the related class automatically creates efficient hash data structures to access the data, and allows the storage of different parameter versions in a single file.

Parameter files typically consist of one or more sections of (rather independent) parameter tables. For example, a force field defines a set of atom types, of Lennard Jones parameters for each of these types, of stretch, bend, and torsion parameters. All these data sets are stored in different sections of a single file.

We represent the whole parameter file as an object of class `Parameters` (see Fig. 6.11). This class uses the `INIFile` class to read the contents of the parameter file, parses the file, and builds internal data structures (several hash maps) to access the contents of the file efficiently. At this level, the data is still stored as tables of strings. The interpretation of the data is carried out by `ParameterSection` and its subclasses. The data a certain section should hold is specified in each class and the user may access this data in a convenient format (usually structs).

For the AMBER force field, this parameter file may look as follows:

```
1  [...]
2  [QuadraticBondStretch]
3  ver:version key:I key:J value:k value:r0 value:comment
4  @unit_k=kcal/mol
5  @unit_r0=Angstrom
6  ;
7  ;
8  ;  Rev  I   J       k         r0      comment
9  ;  --- --- --- ---------- ---------- -----------------------
10    1.0  OW  HW    553.000  0.95720    "! TIP3P water"
```

```
11    1.0  HW  HW    553.000  1.50000    "TIP3P water"
12    1.1  HW  HW    553.000  1.51360    "TIP3P water / corrected"
13    1.0  C   CA    469.000  1.40900    "JCC,7,(1986),230; TYR"
14 [...]
15
16 [LennardJones]
17 ver:version key:I value:R value:epsilon
18 @unit_R=Angstrom
19 @unit_epsilon=kcal/mol
20 @format=RE
21 ;
22 ;
23 ;  Rev  I      R       epsilon    comment
24 ;  --- --- ---------- ---------- -------------------------
25    1.0 C0   1.600000   0.100000 " calcium from parm91.dat"
26    1.0 MG   1.170000   0.100000 " magnesium from parm91.dat"
27 [...]
```

In this example, we see two different sections of the file (marked ”[QuadraticBondStretch]” and ”[LennardJones]”). Comment lines start with ”;” and are ignored by the parser. Lines starting with ”@” are options lines and are automatically parsed into an options object (which is a member of `ParameterSection`). These options are used to store specific properties of the section (*e.g.* the units of the columns or additional format information).

The data format of each section is defined in the first line of a section (lines 3 and 17) and defines the columns of the table. In our case, the Lennard Jones section (used to store van der Waals parameters of the AMBER force field) contains five columns: a version number (column `version`), an atom type (column `I`), the minimum distance (column `R`), the minimum energy (column `epsilon`), and a comment. The `Parameters` class parses all subsequent lines of the section according to this format specification (lines 25–27) and builds a table of strings. The atom type column `I` is marked as a `key` in this table, thus the `Parameters` object also creates a hash map allowing fast access through this column. The first column is a special column: it contains a revision number for each of the lines. Lines with greater revision numbers automatically supercede lines with a lower revision number and identical keys. However, the older revisions of a certain parameter set can be accessed explicitly by specifying an explicit version number. This is an important feature to document the evolution of a parameter set and to access older parameters sets (*e.g.* for the purpose of comparison).

The order of the columns is irrelevant, the subclasses of `ParameterSection` only check whether the parameters they require are available. For example, the `LennardJones` class is satisfied with any table offering a key named `type` and two columns named `R` and `epsilon`. This gives the user the freedom to extend or change the format to suit his purposes. In contrast to column based FORTRAN-style formats, which are used by the majority of software packages, our format is very flexible, readable, and robust.

The `Parameters` class hierarchy gives a very convenient access to arbitrary parameter files and thus simplifies the implementation of the parameter assignment tremendously. Using these classes, parameter assignment becomes nearly trivial, as the following code excerpts from the AMBER force field prove:

```
1  // an instance of LennardJones is used to parse
2  // the corresponding section of the parameter file
3  LennardJones LJ_parameters;
```

```
4   ...
5   // parse the Lennard Jones parameter section
6   bool result = LJ_parameters.extractSection
7     (getForceField()->getParameters(), "LennardJones");
8
9   // check whether we could parse the section correctly
10  if (result == false)
11  {
12    ...  // some error message
13  }
14  else
15  {
16    // iterate over all atoms pairs...
17    for (...)
18    {
19      // ...and determine their type
20      ...
21      Atom::Type I = atom1->getType();
22      Atom::Type J = atom2->getType();
23
24      // retrieve parameters for the atom pair
25      if (LJ_parameters.hasParameters(I, J))
26      {
27        // retrieve the parameters for atom types I and J
28        ...
29        values = LJ_parameters.getParameters(I, J);
30      }
31      else
32      {
33        // we didn't find parameters - error message
34        ....
35      }
36    }
37  }
```

### Energy Minimization

*Energy minimization* (or *geometry optimization*) is one of the two main applications of Molecular Mechanics. The problem is as follows: given a set of molecules with $N$ atoms, determine the set of coordinates $\vec{r_i}$ (x, y, z) for each atom such that the total energy $E(\vec{r_i})$ is minimal. Typically, we are not looking for a global optimum but for a local optimum.

The input sizes range from very few atoms (small molecules, ligands) to large proteins with several thousand atoms. This optimization problem is typically attacked using gradient based techniques like

- steepest descent
- conjugate gradient
- Newton-Raphson
- Quasi-Newton

and a multitude of variants of these methods. The method to be chosen heavily depends on the exact problem case. In some cases (*e.g.* initial relaxation of a constructed structure) the simple steepest descent method is the most efficient one, whereas Newton-Raphson is most

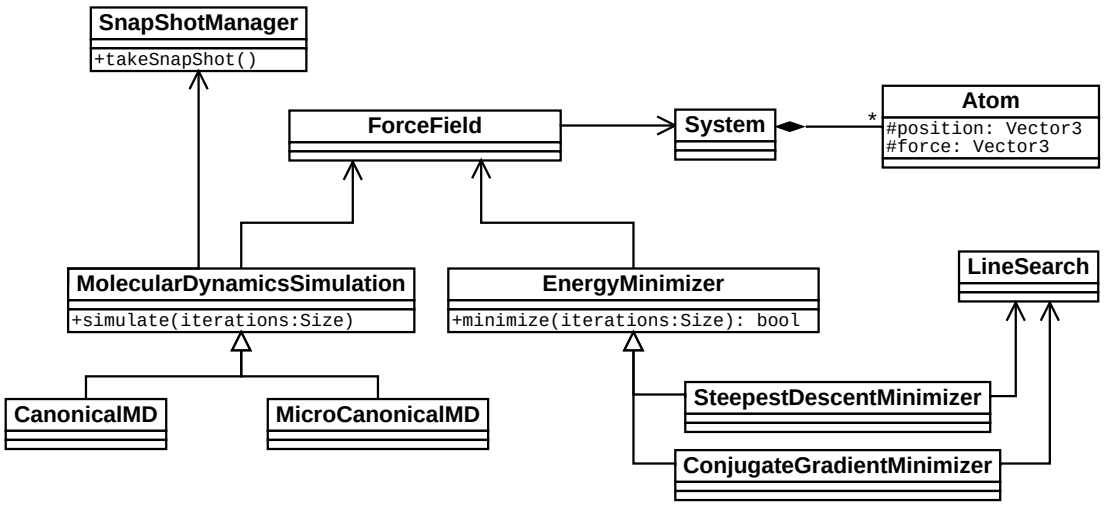



Figure 6.12: *All energy minimization algorithms are derived from* `EnergyMinimizer`*. They make use of the basic line search algorithm implemented in* `LineSearch`*. Similarly, the* `MolecularDynamicsSimulation` *class is the base class of the two implemented ensembles classes (*`CanonicalMD` *and* `MicroCanonicalMD`*).*

efficient for very stringent optimizations, but may lead to memory problems for large problem instances. For a discussion of the different methods and the theoretical background, we refer to [68, 57, 122].

All these algorithms and their variants share large code sections and are based on similar basic algorithms. Hence, we could define a common base class for all energy minimization algorithms (`EnergyMinimizer`). We implemented a steepest descent minimizer and three variants of conjugate gradient minimizers. The design of these classes is shown in Fig. 6.12.

The definition of a common interface to all energy minimizer classes was only possible, because the whole parameter handling (and minimization algorithms require quite some parameters!) could be handled via the `Options` class. We also tried to keep the interface for standard applications as simple as possible. In fact, a typical application (the energy minimization of a small molecule) can be done with just three lines:

```
1 | System S = ...;       // the structure to optimize
2 |
3 | AmberFF FF(S);        // create and setup the forcefield
4 |
5 | ConjugateGradientMinimizer cgm(FF);
6 | cgm.minimize();       // create a minimizer and perform the minimization
```

Nevertheless, it is possible to customize the minimizer to suit most needs. The following example is a bit more demanding and will serve to explain the use of the selection mechanism with the minimizer. A recurring application is the fast relaxation of parts of structures, for example, the optimization of side chains or protein hydrogens. The algorithms used to construct these structures are rather crude and do not check for collisions in the constructed structure, so the next step is the removal of these collisions. Hence, we have to define a subset of atoms that has to be optimized while the remainder should be kept rigid. Furthermore,

we only want to remove the overlaps, so we abort the optimization at a rather large gradient (20 kJ/(mol Å)):

```
1  // a protein with unoptimized hydrogen atoms
2  System p = ...;
3
4  // setup the force field
5  AmberFF FF(p);
6
7  // select the hydrogen atoms only
8  Selector H_select("element(H)");
9  p.apply(H_select);
10
11 // create a minimizer
12 StepestDescentMinimizer sdm(FF);
13
14 // we abort the minimization if the RMS gradient
15 // falls below 20 kJ/(mol A)
16 sdm.options[EnergyMinimizer::MAX_GRADIENT] = 20.0;
17 sdm.minimize();
```

### Molecular Dynamics

The second important application of force fields is the *Molecular Dynamics Simulation* (MD simulation, MDS). In an MDS we apply the rules of Newtionian mechanics to the system described by a Molecular Mechanics force field. By integrating the equations of motion, successive configurations of the system are generated. Thus, we obtain a *trajectory* that describes how the positions of the atoms vary with the time. Again, we have to refer to textbooks for the fundamental theory behind the MDS [38, 68].

MDS trajectories contain a huge amount of interesting information that is closely related to experimentally accessible data (*e.g.* diffusion coefficients, radial distribution functions, *etc.*). So by performing an MD simulation and evaluating the obtained trajectory, we may predict numerous properties of the system we are investigating.

Depending on the specific problem, we have to decide which *ensemble* to choose. The term ensemble stems from Statistical Thermodynamics and basically describes a collection of a very large number of copies of the system whose properties we are investigating. These "copies" differ in their microscopic states, but share the same thermodynamic state (*e.g.* they share the same temperature $T$, volume $V$, and contain the same number $N$ of particles).

There exists a number of different ensembles that differ by the "experimental conditions" of the simulations. In the *canonical ensemble*, the number of particles, temperature, and volume are constant. This corresponds to a set of systems immersed into an infinite bath of temperature $T$. While the number $N$ of particles, volume $V$, and temperature $T$ of each system of the ensemble are identical, their energies $E$ may differ. In contrast to the canonical ensemble, the *microcanonical ensemble* represents an isolated system where $N$, $V$, and $E$ are constant and $T$ varies. These two ensembles are the most common ones, so we implemented two classes that perform MD simulations in a canonical and in a microcanonical ensemble.

The two classes `CanonicalMD` and `MicroCanonicalMD` are both derived from `MolecularDynamics` and only implement the ensemble dependent part of the code. The

handling of the force field and the trajectories (via the `SnapShotManager` class) is implemented in `MolecularDynamicsSimulation` (see Fig. 6.12).

The interface for standard applications is very similar to that of the energy minimizer classes:

```
1 | System S = ....;      // create a system...
2 | AmberFF FF(S);        // ...and a force field.
3 |
4 | CanonicalMD(FF);      // Setup the MD simulation...
5 | FF.simulate(500);     // and simulate 500 time steps.
```

### 6.7.3 Nuclear Magnetic Resonance Spectroscopy

BALL provides a set of classes for NMR shift prediction. The fundamentals of NMR spectroscopy as well as the models for shift prediction, have been explained in depth in Chapter 4. We implemented the shift prediction as a set of shift processors that are responsible for the different contributions to the chemical shift (see Fig. 6.13). Each shift processor class is derived from a common base class (`ShiftModule`). An arbitrary number of these shift modules can be combined into a shift model (class `ShiftModel`). The modules of a shift model can be exchanged and reparameterized at runtime through the `Parameters` class. This class stores the information on the modules to be included and all parameters for these modules in a parameter file (class `INIFile`). Through this mechanism, experiments and reparameterizations of the model are very simple and do not require a recompilation of the code.

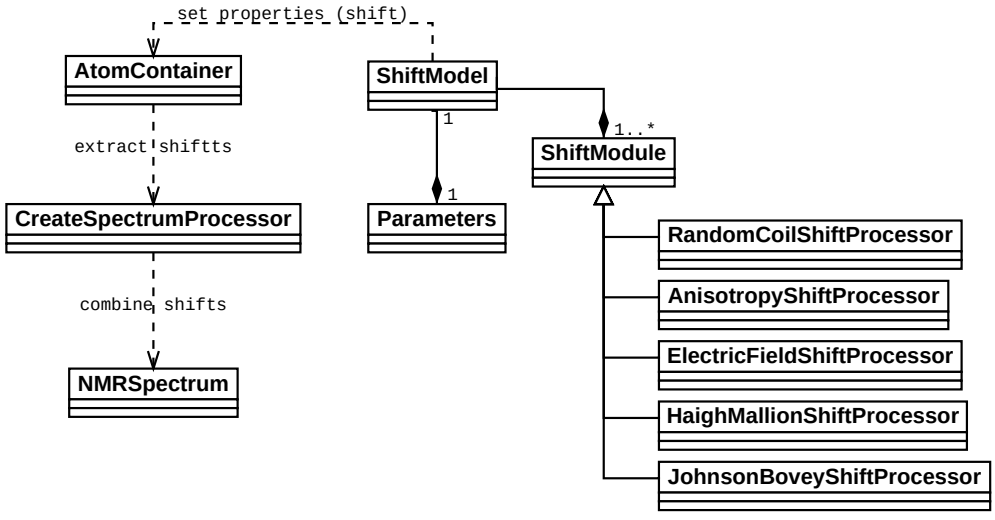



Figure 6.13: *The design of the NMR-related classes in BALL.*

By applying a `ShiftModel` to an `AtomContainer`, the chemical shift of each nucleus is determined by a sequential application of each shift module in the model. The shift is stored as a named property (which can be assigned to any kernel object).

To obtain an NMR spectrum from the assigned chemical shift, we use the `CreateSpectrumProcessor`. This class extracts all nuclei with an assigned chemical shift from an `AtomContainer` object. Then, it discards those atoms that are usually not observable in an NMR spectrum (rapidly exchanging protons), and calculates the spectrum as the linear combination of Lorentzians (one for each proton). This spectrum is then stored in an instance of `NMRSpectrum`. Instances of `NMRSpectrum` can be subtracted to give difference spectra, whose areas were used to rank the docking structures in Chapter 4.

### 6.7.4 Visualization

#### Introduction

Three-dimensional visualization of molecular structures is a very important task in Molecular Modeling – and a very demanding one as well. In fact, the visualization code is roughly one sixth of the total BALL code, so a description of this component has to be even more perfunctory than that of the other components.

Building a framework that allows visualization is very difficult and time-consuming, because it always implies the creation of a graphical user interface (GUI). Writing code that is embedded in a GUI is much more complicated than writing command line applications. One way to circumvent this additional labour is the use of an external viewer. This means, we implement a typical command line-based program. This program then either writes its output to a file or directly connects to an external application that visualizes the results. There is a number of visualization tools available that can be used for this purpose, for example VMD [51] or RasMol [107]. The lack of flexibility is obvious for this approach.

For BALL, we decided to implement a twofold approach:

- a comprehensive and powerful framework for the implementation of GUI based applications
- a stand-alone viewer used as an external visualization tool, which is based on this framework.

The visualization framework is called *BALLVIEW* and the external viewer is called *molview*. Since we did not want to implement our own GUI toolkit, we had to choose one of the existing packages as a basis for the implementation of BALLVIEW. This GUI toolkit should be portable, object-oriented, written in C++, easy to use, and it should provide the means to visualize 3D geometric objects. So QT [102] was the only choice. QT is a portable GUI framework in C++. It implements all the functionality required to write GUI based applications and also provides support for the integration of the OpenGL library [86] for 3D rendering.

BALLVIEW itself consists of two distinct frameworks: VIEW and MOLVIEW. While the VIEW framework implements general visualization of arbitrary objects and data, MOLVIEW provides the specialized models and representations required for the visualization of molecular data structures. The overall architecture of BALLVIEW is illustrated in Fig. 6.14.

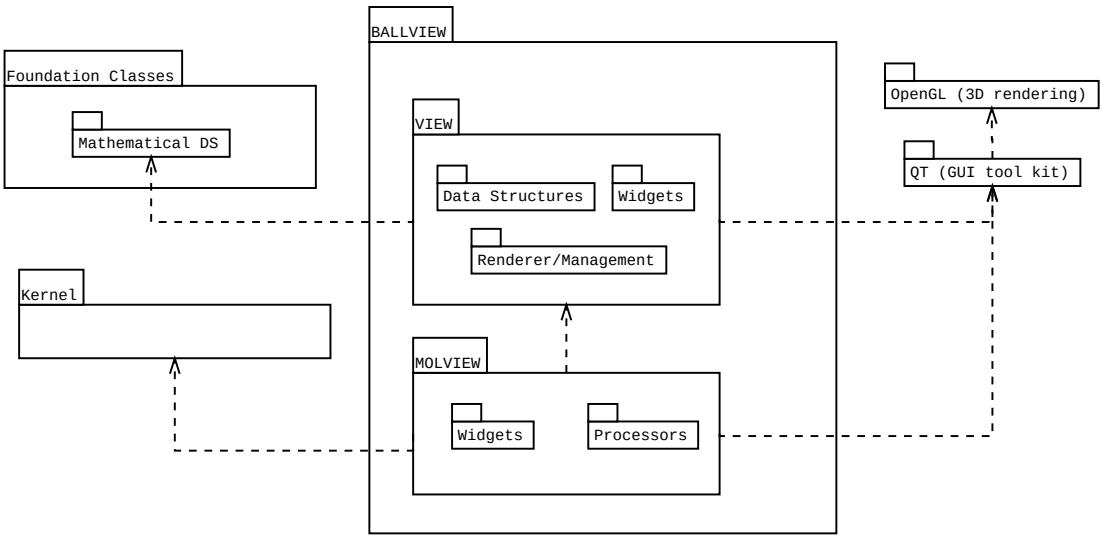



Figure 6.14: *The overall architecture of the visualization component BALLVIEW (see text).*

### The VIEW framework

The VIEW framework provides a number of important basic data structures. First, it defines geometric primitives, like spheres, tubes, meshes, *etc.*, that are required to describe arbitrary geometric objects. Second, it defines data types to handle colors and color tables (*e.g.* to display color-coded gradients or the like). Besides these data structures, it defines a set of classes to access the OpenGL library. These classes build and manage the display lists required for the OpenGL renderer. Finally, there is a number of *widgets* defined that can be used in applications. Widgets in general are graphical elements (*e.g.* buttons, text labels, *etc.*). From the widget classes of the VIEW framework, we will just introduce four important members: the most important widget is the `MainControl` widget. It provides the main window for all applications, including a status bar and a menu bar. This widget is usually the central widget that holds all other widgets. All other widgets are derived from the base class `ModularWidget`. A modular widget can be inserted into the main control and integrates itself automatically. The mechanisms involved are rather complicated, but the effects are simple. The class `ModularWidget` provides a method `initializeWidget` which is called by `MainControl` just before the widget is to be displayed for the first time. In this method, the widget can construct the menu entries it requires, insert icons or labels into the status bar, read its preferences from a global preferences file, or create its own preferences dialog. Each modular widget is thus a complete prefabricated component and these components can be combined like building blocks to form an application. The Rapid Application Development capabilities of this approach will be illustrated by means of an example below.

The third class we will introduce is the `Scene` class. The scene provides three-dimensional interactive visualization of geometric objects. The user can rotate, translate, or zoom into the scene or pick objects by mouse click. This is the key component for three-dimensional

visualization of molecular data structures (see Fig. 6.15).

The fourth class is the `Control`, which is used to display the structures contained in a `Scene` in a tree-like structure and allows the manipulation and inspection of the contents of a scene (see Fig. 6.15).

### The MOLVIEW framework

While the VIEW framework is designed for arbitrary geometric objects, the MOLVIEW framework implements the visualization of molecular data structures. This is achieved through a set of processors. These processors can be applied to arbitrary kernel classes and construct a geometric representation from it. For example, the `AddBallAndStickModel` class constructs a ball-and-stick representation from a molecule, *i.e.* each atom is represented by a sphere and each bond by a cylinder connecting the two atoms. MOLVIEW provides a number of similar processors for other commonly used models, like the van der Waals models (all atoms as spheres), the line model (all bonds as lines), or molecular surfaces. Similar processors are also provided to color the resulting representation according to certain properties, *e.g.* the atom type, the residue name, the charge, *etc.*

The MOLVIEW framework also implements specialized variants of several VIEW widgets that are aware of molecular data structures and several new modular widgets. For example, there are widgets to read molecular data structures from diverse file formats, widgets that implement energy minimizations based on Molecular Mechanics force fields, or a widget that can be applied to add hydrogens to arbitrary structures.

One of the widgets that requires further notice is the `Server` widget. `Server` establishes a network socket in the visualization application. Any client application can now create a `Client` object and connect to that socket. Through the use of object persistence, it is possible to transmit any kernel data structure to the visualization application. The `Server` object receives and deserializes the kernel object and displays it directly if a `Scene` is available. We use this mechanism for our stand-alone visualization tool `molview`, which is explained in more detail below.

### Rapid Application Development using the `ModularWidget` class

As has been mentioned above, the `ModularWidget` class was developed to simplify the rapid construction of applications from some basic building blocks, the modular widgets. Only through the use of the concept of modular widgets, we were able to implement powerful applications with just a few lines of code. The main feature of the `ModularWidget` class is the fact, that each modular widget is self contained and automatically constructs all required connections to other widgets. The internal implementation of these features is quite complicated, so we will just discuss the user interface. When developing applications using QT, one usually assembles a number of widgets and then implements the behavior of these widgets. For this purpose, QT uses the *signal-slot mechanism*. Signals, as well as slots, are functions (usually members). The signal part is called if a certain action occurs, *e.g.* if a button is pressed. By connecting a certain slot to a signal, the slot is called as soon as the signal is *emitted*. For example, pressing the button also calls all connected slots. These slots then implement the required

behavior of the interface. These connections are required for menu entries, to connect slide bars to a scrollable view, *etc.*– the "wiring" of the widgets is usually a very time-consuming and error-prone process.

Modular widgets usually perform this "wiring" automatically. This is achieved by a common, well-defined interface and some internal mechanisms implemented in the `MainControl` class. Each modular widget defines its own menu entries and the related actions. It may also access the application's status bar via methods of `MainControl`. For example, when instantiating a `OpenPDBFileDialog` widget (a widget that reads PDB files), this widget automatically creates an entry in File—Import—PDB in the applications menu bar. Clicking on this menu entry allows the selection of a file, which is then read and a system is created. Now, all relevant widgets have to be informed, that a new system has been read. Using the signal-slot mechanism for this purpose would be very complicated, so we implemented a simple message passing interface. The `OpenPDBFileDialog` widget creates a message stating that a new system has been read and `MainControl` distributes this message (along with a pointer to the system) to all widgets. Certain modular widgets, like the `Scene` class, interpret that type of message: the scene displays the system. Other classes simply ignore the message.

These two mechanisms, automatic signal-slot connections and message passing, create an extensible method for widget communication. Combining modular widgets to an application thus becomes very simple, as the following code example illustrates. We will implement a simple PDB viewer, *i.e.* an application that is able to open and to visualize PDB files (as usual, the include directives have been omitted).

```
1  int main(int argc, char **argv)
2  {
3    // creating mainframe and application
4    QApplication application(argc, argv);
5    MainControl mainframe;
6    application.setMainWidget(&mainframe);
```

The main program first creates a QT application, which manages the GUI control flow. Then, we create an instance of the `MainControl` class and assign it to the main widget of the application, *i.e.* the main control is now the widget that will be displayed in the application window.

```
7    QSplitter* splitter = new QSplitter(&mainframe);
8    mainframe.setCentralWidget(splitter);
9
10   new MolecularControl(splitter);
11   new Scene(splitter);
12
13   new OpenPDBFile(&mainframe);
14   new DisplayProperties(&mainframe);
15   new MolecularProperties(&mainframe);
```

Next, we create a splitter, a widget which lets the user control the size of child widgets by dragging the boundary between the children (the vertical bar in Fig. 6.15). We then create the two child widgets of the splitter: the `MolecularControl`, which displays kernel data structures in a tree-like fashion, and the `Scene`, which renders the molecular structures.

In lines 13–15 we create three additional modular widgets: `OpenPDBFile` is able to read PDB files, `DisplayProperties` creates the dialog that is used to select one of the

different representations for the molecular data structures, and `MolecularProperties` is an (invisible) widget that preprocesses all new kernel data structures, normalizes the atom names and constructs the bonds between the atoms.

```
16 |   // start the application
17 |   mainframe.show();
18 |   return application.exec();
19 | }
```

Finally, in lines 17–18 we just start the application. A screenshot of this program in action is given in Fig. 6.15.

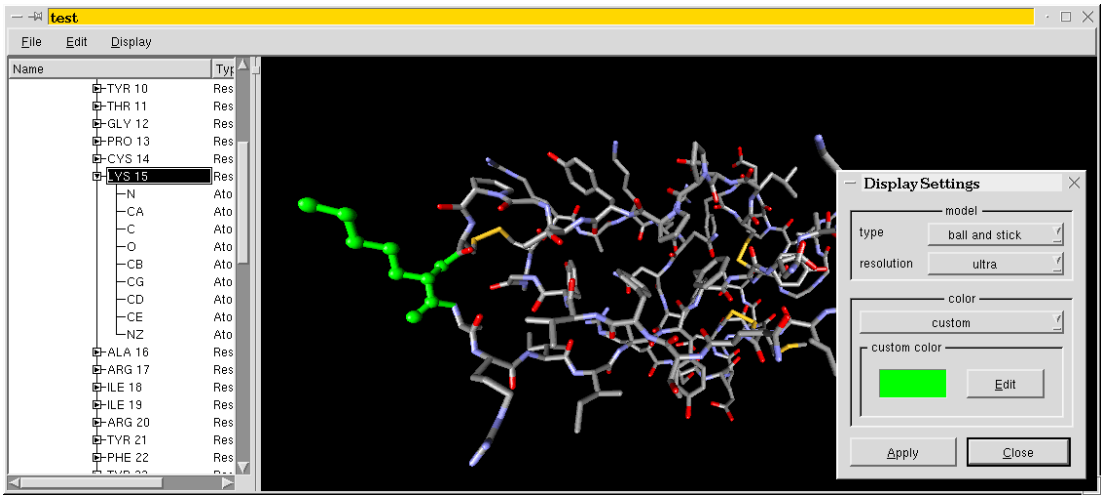



Figure 6.15: *The example code for the MOLVIEW framework in action. There are three widgets visible: the* `MolecularControl` *on the left of the splitter, the* `Scene` *on the right side, and the* `DisplayProperties` *dialog, which creates the non-modal dialog on the right. The* `MolecularControl` *widget permits the inspection, selection, and manipulation of kernel data structures. The* `Scene` *widget displays these data structures according to the representation selected in the* `DisplayProperties` *dialog.*

### The stand-alone viewer `molview`

Even though the implementation of GUI-based visualizations is rather simple, it is often more convenient to write conventional command-line based programs. With the external viewer `molview`, these applications can then visualize arbitrary molecular data structures with just two lines of code:

```
1 |  // some system...
2 |  System S = ....;
3 |
4 |  // create a client for molview running on host "habakuk"
5 |  Client c("habakuk");
6 |
7 |  // send the system and display it on habakuk
8 |  c << S;
```

This code creates a client object for the external viewer `molview`, which is implemented using the VIEW and the MOLVIEW framework). This client connects to a pre-defined port on a given host (in this case the machine `habakuk`). In the last line of the above example, the client creates a persistent representation of the system, which is then sent to the server (the `molview` application) using the network streams of BALL. As soon as the server has received the system, it is displayed on the screen. This approach permits the immediate integration of visualization capabilities into existing applications. Furthermore, it can be used to separate the visualization and the calculations. For example, it is possible to execute a computationally expensive simulation on a powerful computer server and then visualize the result on a graphics workstation.

`molview` also contains functionality to perform energy minimizations using the CHARMM and AMBER force fields, it allows the verification of the structures, the adding of hydrogens, and also an embedded Python interpreter. This embedded interpreter provides additional control over the application, allows the inspection of the data structures, and provides scripting capabilities for `molview`. The design and implementation of the scripting language integration is described in detail in the next section.

## 6.8 Scripting Language Integration

One possibility to speed up Rapid Software Prototpying even further is the integration of scripting languages. Being interpreted languages, scripting languages obviate the time-consuming compile and link stages, thus shortening development times for software prototypes drastically. Interpreted languages are an useful alternative if new applications can be constructed from existing building blocks and do not require the implementation of complex or computationally expensive algorithms. This assumption is valid for the majority of all applications in the field of Molecular Modeling. Especially while optimizing parameters of new methods or adapting existing methods to specific application cases, the shorter turn-around times are very helpful.

Furthermore, it should be possible to port code very easily from C++ to the scripting language and vice versa, thus allowing the prototyping in the scripting language and an easy porting of the final variant to C++ to achieve superior performance.

### 6.8.1 Python

A vast amount of scripting languages has been developed over the time for these specific purposes. The unix world is currently dominated by three scripting languages: Tcl [95], Perl [123], and Python [121]. Each of these languages has been employed as a scripting language in the field of Computational Molecular Biology. For example, there are currently two Open Source projects in progress, namely BioPython [13] and BioPerl [12]. Both projects develop code for applications in the life sciences. However, the focus of these projects is currently on sequence-related algorithms and not on Molecular Modeling.

A detailed discussion of the pros and cons of the three scripting languages would just reiterate well-known arguments. Since all these arguments have been published, we refer to the respective publications [112, 96, 100]. We will just mention a few points that made me favor Python over TCL and Perl. First, Python is the only language that is truly object-oriented

(in the sense that it was developed to be object-oriented, in contrast to the half-hearted support for objects in Perl). Additionally, its readability and intuitive comprehensibility is much better than that of the other two languages. Especially the latter two reasons make it much easier for the average user to learn Python than TCL or Perl. First, we will briefly illustrate some key concepts of Python.

Python[1] has been developed by Guido van Rossum since 1990. It is an object-oriented dynamic language that was designed to be extensible and modular. One reason for its readability is the fact that Python uses indentation to group statements instead of curly braces as many other languages do. To illustrate some of the basic language concepts of Python consider the following example:

```
1 | for n in range(2, 10):
2 |   for x in range(2, n):
3 |     if n % x == 0:
4 |       break
5 |   else:
6 |     print n, 'is a prime number'
```

This simple Python program calculates the prime numbers below 10. The first and obvious point to notice is the block indentation. The compulsory indentation of blocks leaves the programmer no choice – he has to indent the program correctly and the block structure is always obvious. The control structures seem rather familiar to C programmers: `if` conditions, `for` loops – there's just one striking difference: in Python, loop statements may have `else` clauses as well. They are executed if the loop is not aborted via the `break` statement. The `for` loop of Python is quite different from C or Pascal. While `for` loops in C (or C++) can be forced to perform a whole range of tricks, the Python `for` loop has a single purpose: it iterates over all items in a *sequence object*. In the above example, the function `range` creates a sequence of numbers. The `for` loop then iterates over all elements of this sequence. Sequences are of fundamental importance in Python. They come in several flavors, the most important ones being lists (as `range` produces), tuples, and strings. All sequence types may contain objects of arbitrary type since Python is a *weakly typed* language. Python also provides all standard control statements and several other useful data structures (*e.g.* dictionaries).

### 6.8.2 Extending

Python was designed to be easily extensible. This can be done via so-called *extension modules* that are written in either C or C++. These extension modules can implement new object types and functions through an API (Application Programmer's Interface). The API provides nearly complete access to the Python interpreter and provides comfortable functions to integrate C or C++ code into the interpreter.

A BALL extension module should provide access to the major part of BALL's functionality from the Python interpreter and a seamless integration of the BALL objects. The syntax of the extension module should be as similar to the C++ syntax as possible. This allows a fast

[1] Python's not named after some reptile but after *Monty Python's Flying Circus* – the BBC comedy series Guido van Rossum obviously enjoyed.

development of a new method and its refinement in Python and finally – with minor changes only – the implementation of this code in C++ for extensive experiments and production use.

Although both languages are object-oriented, there are some fundamental differences and concepts that make these goals rather difficult to meet. The most important topic is generic programming using templates. Since the creation of template code always requires a run of the C++ compiler, it is not possible to create new instances of templates from the inside of the Python runtime system. Another problem concerns operator overloading: the current release of Python does not yet support compound assignment operators (`+=`, `-=`, *etc.*).

The fact that Python functions may not modify the arguments of a called function if they are *immutable* (as strings are in Python), is problematic as well. Since BALL often passes references of objects to retrieve partial results of a calculation, there are some cases where the interface had to be changed. In these cases, all return values are passed as a Python `tuple`. However, these changes have to be documented and the code cannot be generated automatically, so future versions of BALL will avoid passing non-const object references wherever possible.

The integration of BALL classes into a Python interpreter requires the implementation of so-called *wrapper classes*. Wrapper classes are Python classes that replicate the interface of the C++ class. The methods of these classes call their C++ counterparts and pass the C++ return values back to the Python code. In addition, they have to convert the arguments and return values from C++ objects and types to Python objects and types. Due to the large number of classes in BALL, it is obvious that the generation of the wrapper classes has to be automated. There is actually a number of tools available for this purpose. We had a look at `SWIG` [8], `CXX` [27], and `SIP` [116]. Out of these three, `SIP` was best suited for our purpose. It creates the wrapper classes from slightly modified BALL header files.

Nevertheless, `SIP` did not fulfill all requirements. Therefore, we had to modify it to suit our needs. We added support for C++ operators, meaning that all operators that are supported by Python are automatically converted to equivalent Python methods. Similarly, exceptions are translated to Python exceptions. The latter feature requires all C++ methods to declare the exceptions thrown using the `throw` keyword.

It was also necessary to find replacements for some concepts used in BALL. For example, iterators are not easily ported to Python. The corresponding concept in Python is the iteration over a sequence (via a `for` loop). Hence, I created so-called *extractors*. Extractors are functions that iterate over C++ container objects and assemble a Python sequence containing these objects (or more precisely: reference to these objects). There exists an extractor for each kernel iterator type. For example, the C++ code

```
1 | System S = ...;
2 |
3 | AtomIterator ai = S.beginAtom();
4 | for (; ai != S.endAtom(); ++ai)
5 |   cout << ai->getName() << endl;
```

translates to the following Python code which uses the extractor `atoms`:

```
1 | S = ....
2 |
3 | for atom in atoms(S):
```

```
4 |     print atom.getName()
```

Using these techniques we obtained Python extensions that map the BALL functionality and interface as closely as possible to the corresponding BALL classes and functions. Although there are still some problems to solve (passing of non-const references, several unimplemented operators), the Python extensions have proven to be very helpful when prototyping or debugging small to medium scale applications.

### 6.8.3 Embedding

*Embedding* is quite similar to extending Python. The main difference lies in the main program. When using Python extensions, the main application is the Python interpreter that calls BALL code. When embedding Python, the main application is a BALL program that calls the Python interpreter to execute short code segments. Embedding is required if Python shall be used as a scripting language inside BALL applications (*e.g.* for the molecule viewer `molview`). A further advantage of an embedded interpreter is the ability to inspect and modify the data interactively.

There are two design problems related to Python embedding. The first one is the integration of the interpreter main loop which that to be called as soon as a new code fragment should be executed. Since the C API of Python is rather complicated, this should be well hidden behind a more comfortable C++ interface.

The second issue is somehow more difficult. The interpreter has to have access to the application's data structures. So we need a way to define which object instances of the application may be accessed from the interpreter and vice versa. This is done via the `Embeddable` class. All instances of classes derived from `Embeddable` inherit the members `registerThis` and `unregisterThis`. By calling `registerThis`, a class instance registers with a global objects repository (static members of `Embeddable`). Through the Python bindings of each class, the Python interpreter has access to all registered instances.

The following example shall illustrate the interaction of the classes a bit further (along with Fig. 6.16).

```
 1 | class MyClass
 2 |   : public Embeddable
 3 | {
 4 |   public:
 5 |   BALL_EMBEDDABLE(MyClass)
 6 |
 7 |   MyClass(const String& identifier)
 8 |     : Embeddable(identifier)
 9 |   {
10 |   }
11 |
12 |   void print()
13 |   {
14 |     cout << getIdentifier() << endl;
15 |   }
16 | };
```

This simple class defines just one trivial method (`print`) to print its identifier (which is defined by `Embeddable`). The macro `BALL_EMBEDDABLE` defines the virtual method `reg-`

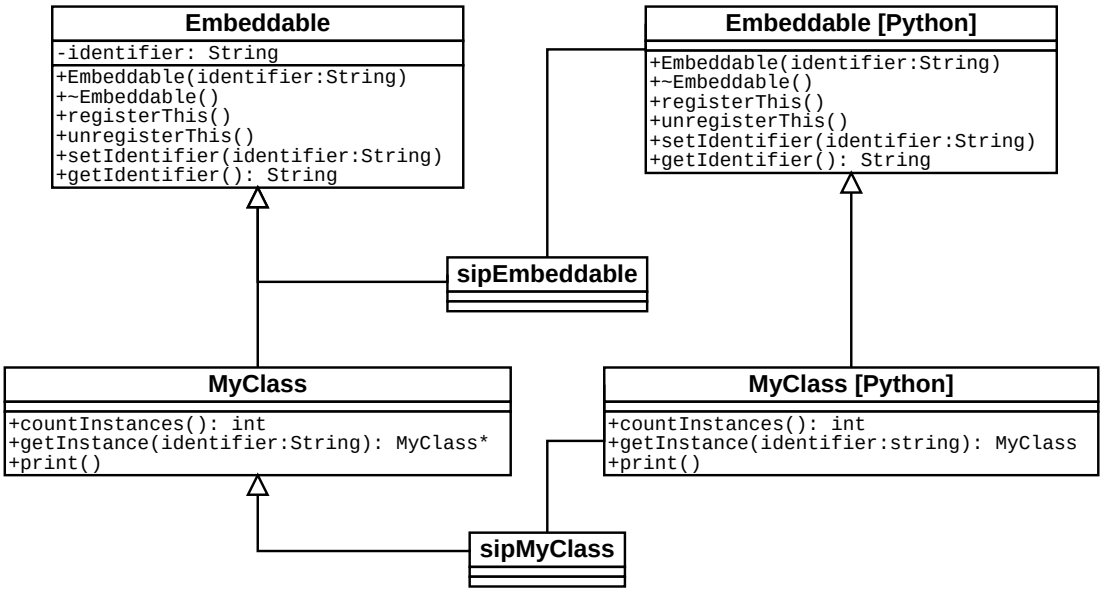



Figure 6.16: *The class* `Embeddable` *is the base class of our class* `MyClass`*. For both classes, wrapper classes (*`sipEmbeddable` *and* `sipMyClass`*) are automatically created for the Python bindings. Through these intermediate classes, Python classes* `Embeddable` *and* `MyClass` *have access to the members of the respective BALL classes.*

`isterThis` that is used to register class instances with the global object repository and the static methods `getInstance` and `countInstances`. The latter methods can be used to access the global instances from Python. The constructor requires a string that is used as a global identifier to access this instance (see below).

Now we create an instance of this class in an application (which should contain an embedded Python interpreter) and register it:

```
1  MyClass my_instance("TESTINSTANCE");
2  my_instance.registerThis();
```

If a Python wrapper was defined for our class and the corresponding module was imported, we can access our instance from the embedded interpreter via its identifier:

```
1  >>> MyClass.countInstances()
2  1
3  >>> instance = MyClass.getInstance("TESTINSTANCE")
4  >>> instance
5  <BALL.MyClass instance at 107cd8>
6  >>> instance.print()
7  TESTINSTANCE
8  >>>
```

The `Embeddable` class defines a very simple but very general interface. Special care has to be taken which methods of these objects can be safely called and who is responsible for the destruction of a certain instance to avoid memory leaks. Obviously, this interface leaves

much room for improvement, but it allows us to embed Python interpreters in applications like `molview` in shortest time and with little overhead.

Chapter 7

# Project Management

This section briefly describes some techniques that were employed to manage the development of BALL and to ensure a high quality of the software. In Section 7.1, we will give a description of the methods to keep track of source code changes, revision management, and portability issues. Section 7.4 describes the integrated documentation, Section 7.5 describes the methods we use for software testing. Finally, in Section 7.6, we will describe the mechanisms required for an automated, portable installation of BALL.

## 7.1 Revision Management

We used the *Concurrent Revision System* (CVS, [24]) to keep track of source code changes. This was especially important in the beginning, when several people were simultaneously working on the core library. CVS keeps all files in a central repository and stores the changes between different versions as context diffs. Even if two or more people are working on a single file in parallel, CVS is usually able to merge the changes into one version of the file – conflicts arise only if the same code lines were changed by different authors. In this case, the respective author is notified of the conflict and has to resolve it manually.

## 7.2 Coding Conventions and Software Metrics

The programming style affects the readability of a program. Hence, to ensure a high code quality and a uniform look of the code, we drafted a set of coding conventions. Coding style is a very personal matter and many different styles exist even among the programmers who worked on BALL. An even larger amount of coding styles has been proposed by various authors (*e.g.* [91, 77, 6]). We drafted a minimal set of non-controversial rules that guarantee a good readability without restricting the individual programmer too much. They are part of a document describing coding conventions for BALL [63]. Part of these rules is also an *orthodox canonical class interface* [21] that ensures that all BALL classes can be assigned, compared, and copy constructed. This common interface ensures that all BALL classes can be used in the same way as typical builtin data types.

However, adhering to these rules is rather inconvenient and compliance with these rules could not be reached without enforcement. Several of the rules in the coding conventions can be verified using *software metrics* (or *style metrics*). Different approaches for style metrics have been proposed in the literature (*e.g.* [10, 49, 19]) for procedural programming languages as well as for OO languages.

The open source software package *CCCC* (C and C++ Code Counter) [72] provides a reasonable choice of different metrics and a WWW-based interface that was easily adoptable

to suit our needs. The metrics we used were *LOC* (*lines of code*), *COM* (*comment lines*), their quotient ($L\_C = \frac{COM}{LOC}$), and McCabe's *cyclomatic complexity* [75].

As with the test builds, an overview of the metrics is created automatically. Its main purpose is to point out design and implementation flaws in the code. The regular inspection of the metrics helps to identify undocumented code portions and too complex (often poorly designed) classes.

## 7.3 Portability

The current ANSI standard for C++ was passed in September 1998 as the ISO/IEC standard 14882 [5]. However, even today lots of C++ compilers are not fully compliant to this standard. This caused a lot of trouble in the earlier stages of the development. Nevertheless, this trouble paid off in the last few months, when more and more compilers were released that are sufficiently close to the ANSI standard to accept our code without major modifications.

## 7.4 Documentation

### 7.4.1 Reference Manual and Tutorial

As has been claimed in Section 6.2, documentation is very important for the user. Our goal was a detailed documentation consistent with the implementation. Therefore, we decided to integrate the documentation into the BALL source code files and use DOC++ [133] to create printed and hypertext manuals from the header files. In this way, the documentation always reflects the current state of the implementation. Unlike other approaches, such as *CWEB* [61], sources documented with DOC++ are ready to be compiled without need of any preprocessing (like `tangle`). DOC++ uses a (high-level) C++ parser to extract the documentation from the header files. Since this parser reads the class declarations as well, DOC++ is able to create information on class hierarchies, member protection, parameters, or return types automatically. The documentation is embedded into special C++ comments starting with '`/**`' or '`///`', hence the code may be compiled directly without modification.

A typical class documentation might look as follows:

```
1  /** Example class.
2      This class is meant as an example.\\
3      Embedding of LaTeX: $a^2 = b^2 + c^2$\\
4      {\bf Definition:}\URL{BALL/test.h}
5      \\
6  */
7  class A
8    : public B
9  {
10   public:
11   /** @name   Constructors and Destructor
12   */
13   //@{
14   /** Default constructor
15   */
16   A();
17   //@}
18
```

```
19      /** @name Accessors
20      */
21      //@{
22
23      /** Example member.
24          This method is useless.
25          @param  x some number
26          @return int the value of the internal attribute divided by {\tt x}
27          @exception DivisionByZero if {\tt x == 0}
28      */
29      int member(float x) const;
30      //@}
31
32      protected:
33      int attribute1;
34    };
```

From this class definition, DOC++ extracts the full documentation in either HTML or LaTeX format. It allows the embedding of LaTeX commands as well as the use of several HTML tags. Even mathematical formulas are rendered using LaTeX and embedded as GIF images into the HTML page (similar to LaTeX2HTML [26]).

Using the special commands '`//@{`' and '`//@}`', the members of a class or sections of the documentation can be arranged hierarchically. The class hierarchy is represented as class graphs in the LaTeX documentation and as a Java applet in the HTML documentation. A screenshot of the resulting HTML documentation, including the class graph, is given in Fig. 7.1 on page 110.

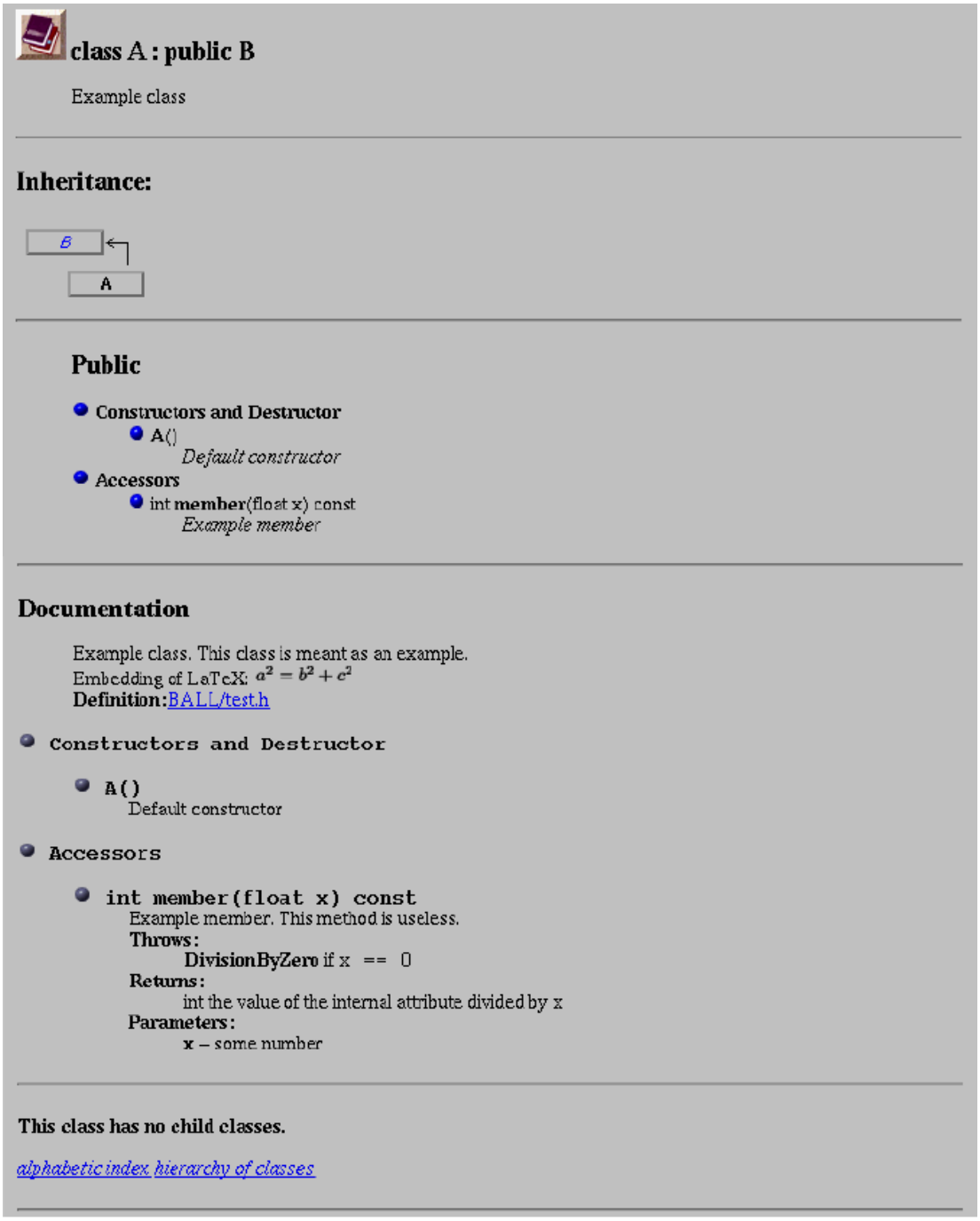
The document resulting from this integrated documentation is the BALL reference manual. It contains detailed descriptions of all classes, their methods and attributes, and references on the algorithms and models used in the implementation. The reference manual is complemented by a tutorial, which introduces the most important concepts and classes by means of selected example applications.

### 7.4.2 FAQs

Another part of the documentation is a *FAQ list* (frequently asked questions). A FAQ list helps the user to resolve common problems. Uwe Brahm implemented the BALL FAQ list similarly to the LEDA FAQ as a Lotus Notes database. We also included the FAQs into the BALL tutorial, because they are most helpful when a user does his first steps with BALL. To keep both versions consistent, the Notes database is exported regularly in ASCII format, converted to LaTeX, and integrated into the LaTeX source code of the tutorial directly.

## 7.5 Testing

Testing is an important step to ensure the correctness of the implemented code and is usually the last step before releasing the code. Since BALL is a quite large project and was already in use while being implemented, we conceived a concept for continuous automatic testing. After introducing some basic terms and definitions in software testing, we will describe this approach. For a further discussion of testing and testing techniques, we refer to [56].



**class A : public B**

Example class

---

**Inheritance:**

B

A

---

**Public**

- **Constructors and Destructor**
  - A()
    *Default constructor*
- **Accessors**
  - int **member**(float x) const
    *Example member*

---

**Documentation**

Example class. This class is meant as an example.
Embedding of LaTeX: $a^2 = b^2 + c^2$
**Definition:** BALL/test.h

- `Constructors and Destructor`
  - `A()`
    Default constructor
- `Accessors`
  - `int member(float x) const`
    Example member. This method is useless.
    **Throws:**
    **DivisionByZero** if `x == 0`
    **Returns:**
    int the value of the internal attribute divided by x
    **Parameters:**
    **x** – some number

---

**This class has no child classes.**

*alphabetic index hierarchy of classes*

---

Figure 7.1: *HTML page created by DOC++ from the example on page 109*

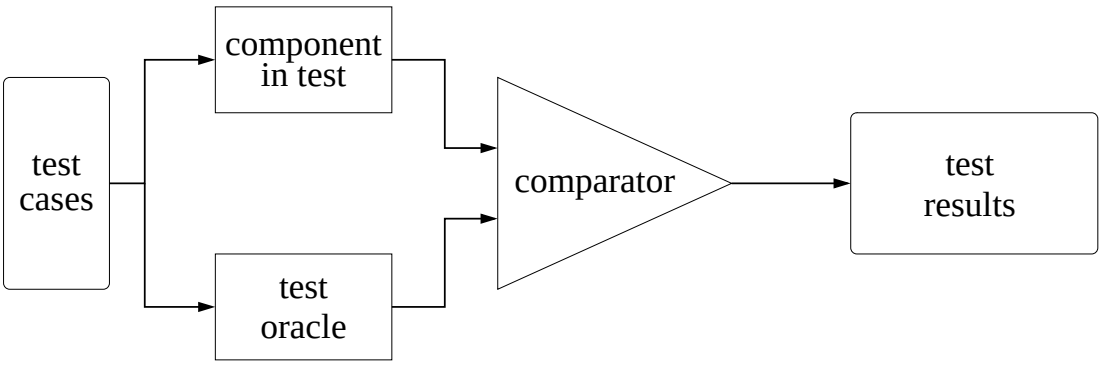



Figure 7.2: *General software testing scheme*

### 7.5.1 Fundamentals

The testing of a software component verifies that its behavior complies with a given specification. For this purpose, a set of *test cases* is created and used as an input to the unit in test (see Fig. 7.2). The output of the component is then compared to the output of a *test oracle*. The test oracle predicts the outcome for each test case based on the specifications. The component complies to the specification if the results of oracle and the component in test are considered equal by the comparator for each test case.

Except for some special cases where formal specifications are created during the design phase (*e.g.* [55]), the generation of the test oracle cannot be automated. Hence, most oracles are simply humans, usually the ones creating the test cases: while creating a set of test cases, the output for each test case is specified as well. The selection of the test cases depends on which approach to testing is used.

There are basically two approaches to testing: *black box testing* (or *functional testing* and *white box testing* (or *structural testing*).

In functional testing, test cases are selected solely on the basis of the specifications of the component. Internals and structure of the component are not considered. Therefore, it is often called black box testing. In contrast to black box testing, white box testing considers the internal structure of the components as well.

### 7.5.2 Testing in BALL

We decided to use simple black box testing for BALL. The introduction of a testing scheme stems from bad experience with the integration of code into the library. Most programmers contributing code to the project tested their code by writing a small number of short programs they chose without much thought. As soon as these test programs performed as the programmer expected (often only on a single platform), the code was submitted to the code repository. Much later, sometimes after the programmer had left the project, serious bugs were detected in the code. Either it did not compile on all platforms due to non portable instructions or header files, or it showed strange behavior for input differing from the original test input, or there were

even syntactically wrong code portions (as might happen for previously uninstantiated template code).

The testing methods we developed for BALL are rather simple, but proved to be rather effective. For each class submitted to the code repository, a test program is required. This test program has to fulfill the following requirements

- it has to test each member of the class
- it has to provide a well-defined interface for automatical testing

The first point is especially important for template classes, where code sections are not compiled if they are not instantiated. Nevertheless, testing of template classes cannot be done satisfactorily: it is not possible to write tests for each possible template parameter. Furthermore, not each method has to be applicable for each template parameter. In these cases, all members are tested; each with a single suitable parameter. This method at least ensures that all code sections are syntactically correct. Then, the methods usually behave correctly for all other suitable template parameters as well.

To ensure a constant software quality, it is not sufficient that a class test once passes on all platforms, but these tests have to be performed on a regular basis on all platforms. This may be illustrated by an example. A test for the BALL class `Object` passed on all platforms. The implementation of `Object` relies on the runtime type identification functions (RTTI), for which no test did exist. When the tests for RTTI were implemented, one of these functions (`isInstanceOf`) was shown to work incorrectly. Instead of returning **true** if the given object was an instance of a certain class, it returned **true** if this object was "a kind of" this class. Thus, it replicated the behavior of the function `isKindOf`. The error was corrected and the test for the RTTI then passed on all platforms. However, a subsequent test of the `Object` class failed. Obviously, in the implementation of `Object` the function `isInstanceOf` was used, where `isKindOf` was meant. So the removal of an error in one component unveiled an error in a completely different component.

This example illustrates two important facts. First, it shows the limits of functional testing: the `Objects` class behaved as specified, although there was an obvious implementation error that could have been found by reviewing the code. Second, it shows the importance of repeated automatic testing of all classes, since changes in one class often affect other classes as well.

### 7.5.3 Test macros

The testing of 460 classes on a dozen platforms has clearly to be automated. This requires a well defined external interface for all test programs. We decided to implement this as a set of macros. The macros implement a program frame for the class tests. The test of a class or component is broken down into several subtests. Each *subtest* verifies the behavior of one method or function and itself consists of several *primitive tests*. In terms of Fig. 7.2, each macro represents a primitive test.

What a typical test program might look like is shown in the following listing:

```
1  #include <BALL/CONCEPT/classTest.h>
2  #include <BALL/DATATYPE/string.h>
3  using namespace BALL;
4
5  // start the class test for class String
6  START_TEST(String,"$Id: implementation.tex,v 1.53 2000/11/13 16:02:22 oliver Exp $")
7
8    String* s;
9
10   // start the first subtest: test the default constructor
11   CHECK(String::String())
12     s = new String;
13
14     // test whether we did get a non zero pointer
15     // this should always pass
16     TEST_NOT_EQUAL(s, 0)
17
18     if (s != 0)
19     {
20       // test whether the string is empty
21       TEST_EQUAL(strlen(s->c_str()), 0)
22     }
23
24   // done with this subtest
25   RESULT
26
27   // further subtests
28   [....]
29
30 // done with all subtests - print the result
31 END_TEST
```

First, the test program includes `classTest.h` which contains the test macro definitions (line 1). Then, it includes the header of the class to be tested – in our example the string class. The first macro to be called is always `START_TEST`. This macro contains the program's main function and performs some initialization steps. Then, an arbitrary number of subtests may follow. Each subtest starts with the macro `CHECK`; its argument is the name of the method or function to be tested. Then an arbitrary number of primitives may follow, terminated by the macro `RESULT` which terminates the subtest. Finally, the test program is terminated by the macro `END_TEST`. When run without arguments, each test program executes all tests and prints `OK` if all test passed, or `FAILED` if any primitive test failed. The test result is also returned via the program's exit code (0 if all test passed, 1 otherwise).

If a test program fails, it may be rerun in verbose mode and then prints information on the test that failed in a standardized format. Further details of the verbose mode will be given below. First, we will give a description of the primitive test macros.

Each of the primitive macros may either pass or fail. If it fails, an internal variable is set to signal a failure of the whole subtest and also the whole test program. The macros usually take two arguments: the result of the "test oracle" (which is usually an expression defined by the human implementing the test) and the result of the component in test. The choice of the macro determines the type of "comparator" used (to speak in the terms of the scheme in Fig. 7.2). There exists a set of macros for different comparison purposes. The most important macro is `TEST_EQUAL`. This macro implements an equality comparison based on the `operator ==`. For example, `TEST_EQUAL(a, 5)` is expanded to something like

```
bool passed = ((a) == (5));
```

This macro assumes that the two expressions given are equality comparable, *i.e.* the `operator ==` has to be defined for the two types.

The opposite behavior is implemented by the `TEST_NOT_EQUAL` macro. This primitive test passes, if the two expressions given as arguments are not equal. This is useful to verify that a method does not return a null pointer and similar cases where only the return value of a failed test may be anticipated.

If the result of a test is a floating point number and deviations from the exact value are to be expected due to different floating point implementations on different platforms, the macro `TEST_REAL_EQUAL` should be used. It verifies whether the absolute value of the difference between the two arguments is below a user-defined upper bound.

It is also possible to test whether the class' exception handling behaves as expected. If a certain method should throw an exception, this can be verified using the `TEST_EXCEPTION` macro. For example, normalizing a vector of length zero throws a `DivisionByZero` exception. A primitive test to verify this behavior might look like this:

```
1 | Vector3 null(0, 0, 0);
2 | TEST_EXCEPTION(Exception::DivisionByZero, null.normalize())
```

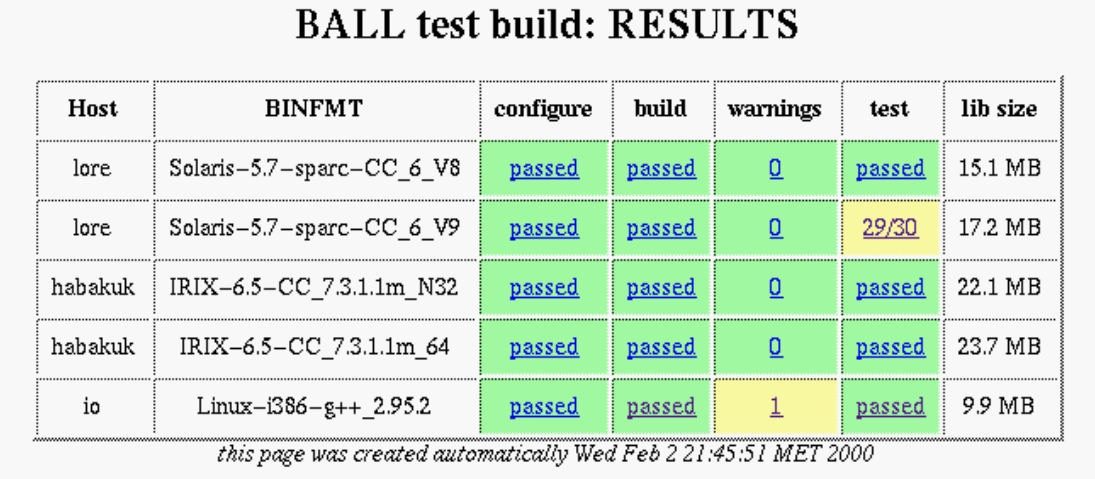
This test passes, if `null.normalize()` really throws the exception, which is then caught by the test macro. Unexpected exceptions are caught automatically by the try/catch block implemented in `CHECK` and `RESULT` if they occur inside a subtest or by the try/catch block implemented in `START_TEST` and `END_TEST` if they are thrown outside a subtest. In any case, the test program fails and a diagnostic message specifying the kind of exception thrown is printed in verbose mode.

Finally, the `TEST_FILE` macro is suitable to compare the output written to a file. For example, this macro is used to verify the PDB file class which is used to import or export molecular structures in the PDB format. In principle, this should be rather simple: read a `System` from disk, write it to a temporary file in PDB format, and compare this file with the original file. Unfortunately, this approach fails for the very first line of the PDB format: this line contains the date the file was created. This problem is solved by allowing regular expressions for the files to be compared. The two files are compared line-wise. If a line in the template file (the "oracle output") starts with a `'/'`, the rest of the line is interpreted as a regular expression and the corresponding line in the temporary file has to match this expression. In our example of the PDB class test, the first few lines of the test output might look like this:

```
1 | HEADER    BPTI                                  6-DEC-99
2 | SEQRES   1     58  ARG PRO ASP PHE CYS LEU GLU PRO PRO TYR THR GLY PRO
3 | SEQRES   2     58  CYS LYS ALA ARG ILE ILE ARG TYR PHE TYR ASN ALA LYS
4 | SEQRES   3     58  ALA GLY LEU CYS GLN THR PHE VAL TYR GLY GLY CYS ARG
5 | SEQRES   4     58  ALA LYS ARG ASN ASN PHE LYS SER ALA GLU ASP CYS MET
6 | SEQRES   5     58  ARG THR CYS GLY GLY ALA
```

The corresponding lines in the template file are:

```
1 | /HEADER BPTI                                  [ 1-3][0-9]-[A-Z][A-Z][A-Z]-[ 0-9][0-9]
2 | SEQRES   1     58  ARG PRO ASP PHE CYS LEU GLU PRO PRO TYR THR GLY PRO
3 | SEQRES   2     58  CYS LYS ALA ARG ILE ILE ARG TYR PHE TYR ASN ALA LYS
4 | SEQRES   3     58  ALA GLY LEU CYS GLN THR PHE VAL TYR GLY GLY CYS ARG
```



**BALL test build: RESULTS**

| Host | BINFMT | configure | build | warnings | test | lib size |
|---|---|---|---|---|---|---|
| lore | Solaris-5.7-sparc-CC_6_V8 | passed | passed | 0 | passed | 15.1 MB |
| lore | Solaris-5.7-sparc-CC_6_V9 | passed | passed | 0 | 29/30 | 17.2 MB |
| habakuk | IRIX-6.5-CC_7.3.1.1m_N32 | passed | passed | 0 | passed | 22.1 MB |
| habakuk | IRIX-6.5-CC_7.3.1.1m_64 | passed | passed | 0 | passed | 23.7 MB |
| io | Linux-i386-g++_2.95.2 | passed | passed | 1 | passed | 9.9 MB |

*this page was created automatically Wed Feb 2 21:45:51 MET 2000*

Figure 7.3: *Web interface for automatic testing: Overview page*

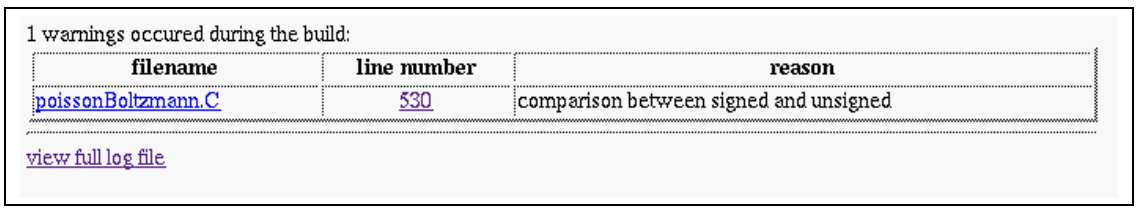
```
5 SEQRES   4     58  ALA LYS ARG ASN ASN PHE LYS SER ALA GLU ASP CYS MET
6 SEQRES   5     58  ARG THR CYS GLY GLY ALA
```

The regular expression in the first line matches all possible dates as defined in the PDB format.

### 7.5.4 Automatic Test Builds

Using these macros simplifies the implementation of test programs and creates a well defined interface. Using this interface, a suite of shell scripts was developed that performs a automated test build on all available platforms. These platforms are specified in a configuration file. The script then extracts the current version of BALL from the CVS repository, runs `configure`, builds the shared library, and executes all class tests. Finally, web pages are created that show the results of the test builds for all platforms.

The overview page might look like the one in Fig. 7.3. It shows the result for the execution of configure, the build of the library, the build of the test, and the execution of all tests for five different platforms. Yellow table entries indicate problems, green ones indicate that the corresponding step was successful. In the above example, one test failed and the compilation of the library caused a compiler warning. Compiler warnings and errors are extracted from the log files and are shown by following the link in the corresponding table entry:



1 warnings occured during the build:

| filename | line number | reason |
|---|---|---|
| poissonBoltzmann.C | 530 | comparison between signed and unsigned |

view full log file

By following the links for each error, the source code causing the warning can be directly inspected. The line causing the error is highlighted in red:

**SOLVATION/poissonBoltzmann.C:530:**

comparison between signed and unsigned

responsible for this file: oliver

```
520     eps_grid = new PointGrid<float>(lower_, upper_, spacing_);
521
522     // now assign the correct dielectric constant to each grid point in eps_grid
523     long inside_points;
524     long outside_points;
525     inside_points   = 0;
526     outside_points  = 0;
527
528     // loop variable
529     Index i;
530     for (i = 0; i < eps_grid->getSize(); i++)
531     {
532       if ((*SES_grid)[i] == 0)
533       {
534         outside_points++;
535         (*eps_grid)[i] = solvent_dielectric_constant;
536       } else {
537         inside_points++;
538         (*eps_grid)[i] = solute_dielectric_constant;
539       }
540     }
```

This web interface greatly simplifies the maintenance of the code on all platforms since it immediately reveals any problems with new code. Since a test build is performed every night on all platforms, committing faulty code to the repository becomes obvious the latest by the next day. Using the revision control system, a controlled roll back of the software to the last version that ran on all platforms can be obtained at any time.

## 7.6 Installation and Configuration

An important aspect of our design goal *ease of use* is a simple and smooth installation on all required platforms. When we examined existing software packages before we started the implementation of BALL, installation was a continuous source of frustration. So our goal was to make the installation of BALL as simple as possible. This is a truly demanding task. First, all platforms are different, all compilers take different options, different versions of software or operating system require special workarounds, header files differ from platform to platform, and finally every local installation has its own peculiarities.

*GNU autoconf* [37] is a portable and extensible tool for the installation of software packets. It uses a very limited set of standard unix tools (`sh`, `ar`, `sed`, *etc.*) to gather all information on the system that is required to automatically build and install the libraries. So if everything works as expected, a typical installation would only require to run the configuration script `configure` and the build the library with the command `make`. `configure` is a rather complex shell script (about 7500 lines long) that identifies the operating system and the compiler and then searches for the required additional software (*e.g.* libraries and header files).

Contrary to installation routines that only provide platform dependent configuration files (*i.e.* a set of files for each combination of operating system and compiler that is supported), `configure` is based on tests to identify the platforms capabilities. For example, instead of stating that the GNU compiler does not provide the ANSI class `numerical_limits`, the `configure` script runs a test program that determines whether the `numerical_limits` class is available. This approach is clearly superior, as it requires very little effort when porting the library to a new platform. Most of the porting is done by the `configure` script when testing for required features.

Writing these tests in a portable and general way is nevertheless rather tricky and requires a good knowledge of unix systems and their differences. But this effort usually leads to a very convenient installation, as has been shown in our alpha test experiences. Even if the installation fails, configure provides mechanisms to solve these problems from a distance. For example, the results of all tests are written to a log file. Usually, by mailing the output of the `configure` script and this log file, most installation problems can be resolved.

Chapter 8

# Programming with BALL

In this chapter, we will illustrate the use of BALL for Rapid Software Prototyping by means of a simple example: the minimization of a protein in vacuum. This example is originally taken from the tutorial of the Molecular Modeling package AMBER [22]. To illustrate the differences between a conventional package (AMBER) and BALL, we will contrast the original AMBER code directly with the BALL code.

The tutorial example performs a conjugate gradient minimization of the protein BPTI in vacuum after adding the required hydrogen atoms. Since AMBER is a mere collection of FORTRAN programs, the code shown here is mainly shell code and input files for the different FORTRAN programs.

### Preparation of the structures

In a first step, we have to add the hydrogen atoms to the BPTI structure, since they are usually not contained in a PDB file. Using AMBER, this is accomplished by the `protonate` program:

```
1  protonate -d amber41/dat/PROTON_INFO < pti.pdb > ptiH.pdb
```

The call to `protonate` reads the PDB file, adds all missing hydrogens, and stores the information to the file `ptiH.pdb`. In BALL, the same task is accomplished by the `add_hydrogens` member of the fragment database after reading the PDB file:

```
1  PDBFile infile("pti.pdb");
2  System S;
3
4  infile >> S;
5
6  FragmentDB frag_db;
7  S.apply(frag_db.add_hydrogens);
```

### Setting Up the Force Field

Using AMBER, the next step is to establish the topology of the force field. To achieve this, the program `link` is called which reads information on the bonds from a file `dat/db4` in the AMBER data directory (set in the environment variable `$AMBERHOME`. Prior to the call to `link`, an input file has to be constructed that contains the sequence information of the protein and the necessary information on sulfur bridges:

```
1  cat <<eof >lnkin
2   bpti
3
```

```
 4
 5  DU
 6      0    0    0    0    0
 7   bpti
 8  P   1    0    1    3    1
 9  ARG 2PRO  ASP  PHE  CYX  LEU  GLU  PRO  PRO  TYR  THR  GLY
10  PRO  CYX  LYS  ALA  ARG  ILE  ILE  ARG  TYR  PHE  TYR  ASN
11  ALA  LYS  ALA  GLY  LEU  CYX  GLN  THR  PHE  VAL  TYR  GLY
12  GLY  CYX  ARG  ALA  LYS  ARG  ASN  ASN  PHE  LYS  SER  ALA
13  GLU  ASP  CYX  MET  ARG  THR  CYX  GLY  GLY  ALA
14
15      5   55SG  SG      0
16     14   38SG  SG      0
17     30   51SG  SG      0
18
19  QUIT
20  eof
21  #
22  link -i lnkin -o lnkout -p  $AMBERHOME/dat/db4.dat
23  /bin/rm lnkin
```

The first seven lines contain various configuration flags, that are usually not changed. If changes are done here, one has to be very careful, as the input format is strictly column-oriented (as the input formats of most AMBER input files as well), so the number of blanks before each number is crucial and blanks and empty lines have to be counted carefully to avoid unwanted behavior. The next five lines contain the sequence information that has to be extracted from the PDB file manually. Lines 15–17 contain information on the three sulfur bridges of the structure. Again, this information can be extracted from the PDB file. The last two lines finally call link and create the topology information in file `lnkout` and in the file `linkbin`, which is used in the next step by the `edit` program.

The same code in BALL looks as follows:

```
8 |  S.apply(frag_db.build_bonds);
```

This code constructs all bonds, including the sulfur bridges from the topological information contained in the fragment database.

In the next step, AMBER checks the structure of the PDB file for consistency and writes a set of starting coordinates. This step is carried out using the `edit` program:

```
 1  cat <<eof > edtin
 2  bpti
 3      0    0    0    0
 4  XYZ
 5  OMIT
 6  XRAY
 7      0    0    0    0    0
 8  QUIT
 9  eof
10  #
11  edit -i edtin -o edtout -pi ptiH.pdb
12  /bin/rm edtin
```

This script reads the PDB file from `ptiH.pdb`, checks it for consistency, and writes starting coordinates for all atoms to `edtout`. It also creates an additional file (`edtbin`) that contains

information on the coordinates and is used in the next step by `parm`.

A similar functionality (although much more sophisticated) is contained in the residue checker class of BALL. This class exploits the topological and geometric information contained in the fragment database to verify the residues it is applied to. Upon failure, it emits detailed warning messages.

```
 9 |  ResidueChecker checker(frag_db);
10 |  S.apply(checker);
```

In the next step, AMBER reads the information from `edtbin` and the force field parameter file `parm91.dat`. From these files, it extracts all relevant information on the coordinates, the atom types, and the parameters required for the force field calculation (*e.g.* force constants, bond lengths, *etc.*).

```
cat <<eof >prmin
 name of system
BIN FOR STDA

    0

eof
#
parm -i prmin -o prmout  -f $AMBERHOME/dat/parm91.dat
/bin/rm prmin
```

### Running the Minimization

The minimiztion itself is accomplished by the program `sander`, which reads the input from `prmtop` and `prmcrd` created in the last step:

```
cat << eof > minin
#  200 steps of minimization:
 &cntrl
    maxcyc=200, imin=1, cut=12.0, nsnb=20, idiel=0, scee=2.0, ntpr=10,
 &end
eof
sander -i minin -o pti_min.log -c prmcrd -r pti_min.xyz
/bin/rm minin
```

This script performs a conjugate gradient minimization for a maximum of 200 steps (given as the value for `maxcyc`). The fact that a full minimization is performed, is specified via the value of `imin`: a value of 1 means a minimization, while a value of 0 starts a Molecular Dynamics simulation. The remaining parameters in the fourth line set the parameters for the nonbonded cutoff (`cut`) to 12 Å, and choose a distance-dependent dielectric constant for the simulation (`idiel=0`). The current energy during the minimization will be printed every ten steps (`nptr=10`) and the pair list will be updated every 20 steps (`nsnb=20`). The parameter `scee` specifies a scaling factor for the electrostatic contribution of the force field. The resulting structure is written to the file `pti_min.xyz`.

In BALL, the code of the last two steps corresponds to the creation of a force field object and a conjugate gradient minimizer object:

```
11  AmberFF force_field;
12  force_field.options[AmberFF::Option::FILENAME] = "Amber/amber91.ini";
13  force_field.options[AmberFF::Option::NONBONDED_CUTOFF] = 12.0;
14  force_field.options[AmberFF::Option::DISTANCE_DEPENDENT_DIELECTRIC] = true;
15  force_field.setUpdateFrequency(20);
16  force_field.setup(S);
17
18  ConjugateGradientMinimizer minimizer(force_field);
19  minimizer.options[EnergyMinimizer::Option::ENERGY_OUTPUT_FREQUENCY] = 10;
20
21  minimizer.minimize(200);
22
23  PDBFile outfile("pti_min.pdb", File::OUT);
24  outfile << S;
```

The above code creates a force field object, assigns the correct options (for the filename, the nonbonded cutoff, and the distance-dependent electrostatics) and sets the pair list update frequency to the desired value. A call to `setup` then sets up all force field parameters and internal data structures. We then create a conjugate gradient minimizer in line 18, set its energy output frequency to the desired value, and finally perform a maximum of 200 minimization steps (line 21). The resulting structure is then written to a PDB file in the last two lines.

As can be seen from this example, BALL can significantly reduce the amount of code and time required in the field of Molecular Modeling. When compared to conventional software packages, BALL excels especially in the readability of the code and the robustness. Especially the preparation of the input files for FORTRAN programs is often very error-prone due to the column-based style they require and the diversity of format definitions. BALL, in contrast, provides a much more convenient and comprehensible interface.

Chapter 9

# Outlook

With BALL, we implemented the first object-oriented framework for Rapid Software Prototyping in the field of Molecular Modeling. It differs from functionally related packages in its careful design, the use of modern software engineering techniques, and the comprehensive functionality. In this section, we will briefly point out some of the current problems and future directions for improvement and development.

Most of the current problems are related to the sheer size of the project: BALL contains about 460 different classes implemented with 270,000 lines of code. An additional 30,000 lines of code implement the class tests. Keeping this amount of code consistent is currently one of the major challenges in the development of BALL.

Also the testing of the classes is not yet complete. There are tests implemented for roughly one third of the classes, so a large number of bugs still awaits discovery. Similarly, there is still a number of classes with lacking or incomplete documentation. These gaps have to be filled in the near future as well.

During the alpha test phase, we received extensive feedback from the users. The majority of these mails was concerned with requests for additional functionality. It is certainly impossible to satisfy all the wishes, but there are some key areas that are obviously of general interest. The first field is Molecular Mechanics. Besides the wishes for additional force fields, there is also need for harmonic constraints, and superior optimization methods (quasi-Newton methods, internal coordinate optimization). Second, the visualization component still requires a number of improvements. Besides additional visualization techniques (*e.g.* backbone and secondary structure representations of proteins or mapping of properties onto molecular surfaces), we also have to speed up the the visualization component. The Python extensions/embedding, although already very useful, does not yet cover all BALL classes and several technical problems related to the automatic generation of the wrapper classes remain to be solved.

Since we intend BALL to become a widely used tool, licenses are free of charge for research and teaching, although commercial licenses and support are available as well (through our partner Algorithmic Solutions Software GmbH). In that way, we hope to resolve some of the topics mentioned above in cooperation with other groups that are working in the field of Molecular Modeling.

# Part IV

# Conclusion

In this thesis, we have presented new approaches for the protein docking problem as well as a comprehensive framework in C++ for rapid software prototyping in the field of Molecular Modeling.

### Semi-flexible docking

First, we presented two new techniques for semi-flexible protein-protein docking. Based on a rigid-body docking algorithm, we developed methods to demangle the amino acid side chains in the protein-protein interface. This side chain placement problem can be reduced to a combinatorial optimization problem: identify the set of side chain conformations that yields the lowest energy. The first technique is a so-called multi greedy approach, *i.e.* we search the enumeration tree of all possible side chain conformations and avoid combinatorial explosion by limiting the number of nodes in each layer of the tree to a constant number. The second approach is a branch-&-cut algorithm based on the ILP formulation of the problem. We identified several classes of facet-defining inequalities and devised a separation algorithm for a subclass of these inequalities. With the algorithm implemented using BALL, LEDA, and CPLEX, we could efficiently solve the side chain placement problem optimally. The suboptimal solutions produced by the greedy algorithm were sufficiently close to the optimal solutions to give very similar results in protein docking: in a test set of three protease-inhibitor complexes, both methods were able to correctly demangle the side chains. The resulting complex structures were physically meaningful and therefore gave a good ranking in the final energetic evaluation step, so both approaches were able to predict the true complex structure for each of the test cases. Future improvements of the methods could employ branch-&-price techniques on the algorithmic side or improved energy models and more detailed rotamer libraries on the physicochemical side.

### Use of NMR data for protein docking

Since the energetic evaluation of the docking is a demanding and unsolved problem, we also tried to identify other possibilities to separate true and false positive complex structures. We developed an approach that performs this separation based on experimental data. This approach exploits the structural information contained in nuclear magnetic resonance spectra of the complex. Although there have been previous approaches in protein-ligand docking and protein-protein docking that incorporate NMR data in the form of NOE constraints, our method is the first method to integrate *unmodified* experimental data into the docking algorithm. We developed models and methods to predict the $^{1}$H-NMR spectra of all tentative complex structures generated by a rigid-body docking algorithm. These spectra were then directly compared with the experimental NMR spectrum and the difference area between the two spectra was used to rank the docking candidates. This approach worked quite well for a set of test cases and was even able to identify the correct complex structure for a very difficult protein-peptide docking example, where energy-based scoring functions were completely unable to predict the true complex structure. This new technique should be applied to a larger set of test cases. The relevant data is usually not publicly available, but we hope to get access to that data through cooperations with NMR research groups. The prediction and comparison of the spectra still

has lots of room for improvements, *e.g.* to model hydrogen bonding effects. The technique is applicable to related problems as well, and thus we hope to be able to speed up NMR-based structure elucidation and to transfer our method to the problem of protein structure prediction.

### BALL

The above mentioned methods were developed using the BALL framework. BALL is the first object-oriented framework for rapid software prototyping in molecular modeling. It has been carefully designed to be robust, portable, efficient, and extensible. Besides fundamental data structures, BALL provides functionality for a number of key areas in Molecular Modeling. We implemented several molecular mechanics force fields, energy minimization algorithms, molecular dynamics simulations, solvation methods, protein motif search and mapping, three-dimensional visualization, file import/export, and NMR shift prediction. BALL consists of approximately 460 different classes and 270,000 lines of code. To improve the library's rapid prototyping capabilities, we embedded the BALL classes into the object-oriented scripting language Python. After an application has been developed in Python, it is very simple to port it to C++ for production purposes. The BALL library is available for a number of operating systems and compilers and should be easily portable, as it is implemented in ANSI C++. BALL has been used in several labs in Europe and North America in the current alpha test phase and we hope to release the first beta release to a larger audience soon. There still remain a number of problems to be addressed, namely the documentation, which needs improvements and the test suite, which does not yet cover all BALL classes. As soon as these problems have been resolved, we will extend the functionality of BALL, especially in the fields of Molecular Modeling and Protein Engineering. We intend for BALL to become a public repository for reliable data structures and algorithms in the field of Molecular Modeling.

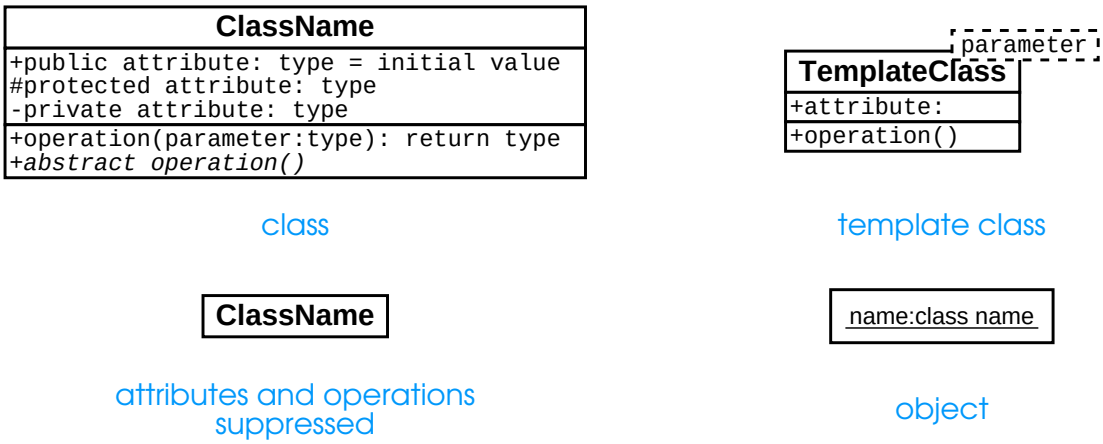
Appendix A

# UML Notation

This work uses the Unified Modeling Language (UML) to represent design details of BALL. The class diagrams were drawn according to the UML 1.3 standard [90]. A short introduction to this notation can be found in the book by Martin Fowler [36]. A rather comprehensive reference is the book by Rumbaugh, Jacobson, and Booch [103] who are the original designers of the UML. This appendix gives a short overview of the notation used in Part III. The most important concepts of the UML are also explained in the glossary.

## Classes and Objects

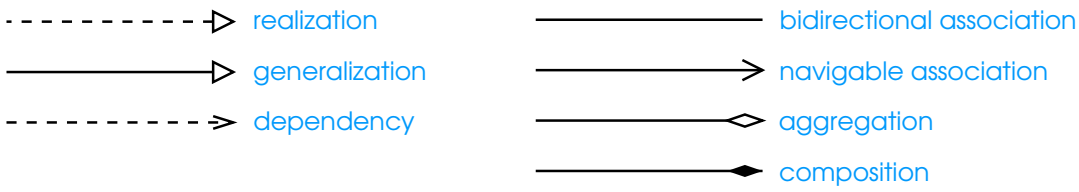
A central concept in UML is the *class*. A class may contain *attributes* and *operations*. The notation for a class is a rectangle with the class name on top. The lower part of the rectangle usually contains two sections with the attributes and the operations. Public attributes and operations are prefixed by "+", protected ones by "#", and private ones by a "-" sign. Abstract operations are typeset in italics. Template classes are marked by an additional box at their upper right corner that contains the template parameter. Objects (*i.e.* instances of classes) are distinguished from classes by the underlined object name and class name.




## Relationships

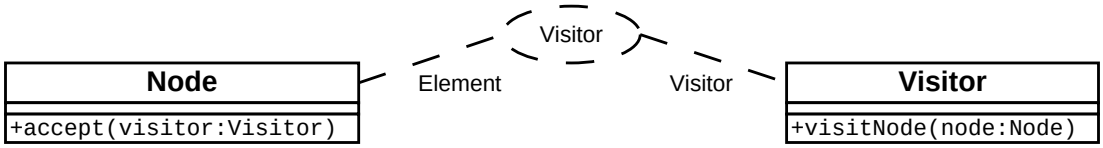
Relationships between objects and classes are represented by different kinds of lines and arrows. *Associations* are represented as solid lines and may be adorned with role names, association names, and multiplicities (just to name the most important decorations). Associations without arrows are bidirectional, whereas associations with an arrow are unidirectional. They

are only *navigable* in the direction of the arrow. *Generalization* is expressed as solid line from the specialized class to the base class with a hollow triangle at the end of the base class. Lines with hollow or filled diamonds are used to express *aggregation* and *composition*. They are usually decorated with multiplicities and rolenames. A *realization* is the relationship between a specification and its implementation. It is represented as a dashed line with a hollow triangle at the end of the specification. A dashed line with an arrow represents a *dependency* between two elements. It is usually adorned with the name of the dependency.




## Design Patterns

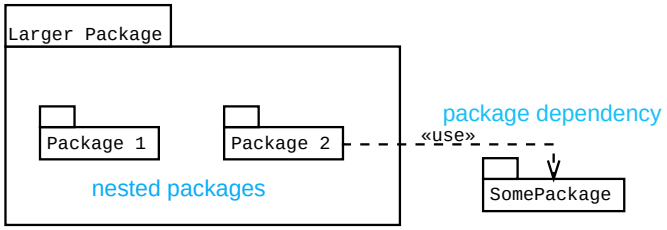
Design patterns are usually modeled as a *collaboration*. They are shown as dashed ellipes containing the name of the pattern and dashed lines from the ellipse to each participating class.




The above example shows part of the visitor design pattern and two of its participants: the *Visitor* class and the *Node* class. The dashed ellipse is labeled with the name of the pattern and the bindings of the collaboration are labeled with the name of the role in the pattern.

## Packages

UML allows the grouping of model elements and diagrams with so-called *packages*. Packages may be nested within other packages, thus creating hierarchical groupings. Packages are drawn as a rectangle with a tab attached on the upper left corner. This tab contains the name of the package, if its contents are shown. If the contents of a package are not shown, the tab usually remains empty and the package name is put inside the package rectangle. Relationships between packages are drawn in the usual manner. Dependencies between packages indicate that dependencies exist at least between some elements from each of the packages.



## Appendix B

# Curriculum Vitae

Oliver Kohlbacher
Johann-Strauß-Straße 13
66793 Saarwellingen

**Date of Birth** 29.08.1971.

**Citizenship** German.

**Education**

MAX-PLANCK-INSTITUT FÜR INFORMATIK — SAARBRÜCKEN
1996–2000
PhD student.

UNIVERSITÄT DES SAARLANDES — SAARBRÜCKEN
1990–1996
Diplom (≈ Master's degree) in Physical Chemistry in 1996. Thesis: *ab-initio calculations of silicate clusters: contributions to the simulation of adsorption processes on* $SiO_2$, advisor: Prof. Dr. Hans Dieter Breuer.

MAX-PLANCK-GYMNASIUM — SAARLOUIS
1990
Abitur (≈ Highschool Diploma).

**Work and Teaching Experience**

I-TRONIX KG — SAARBRÜCKEN
1999 – present
Geschäftsführer (CEO).

UNIVERSITÄT DES SAARLANDES — SAARBRÜCKEN
1992–1998
Teaching assistant for the following lectures:
- Organic Analytics,
- Computer Science IV,
- Computational Molecular Biology.

MAX-PLANCK-INSTITUT FÜR INFORMATIK SAARBRÜCKEN
1992–1996
System administration in the computer service group.

**Awards**

2000
Heinz Billing Award 2000 for the Advancement of Scientific Computation (for the development of BALL)

**Publications**

**Journal Publications**

1. KOHLBACHER, O., AND LENHOF, H.-P. BALL - Rapid Software Prototyping in Computational Molecular Biology. *Bioinformatics* (2000). (in press).

**Book Chapters**

2. KOHLBACHER, O. BALL – A Framework for Rapid Application Development in Molecular Modeling. In *Beiträge zum Heinz-Billing-Preis 2000*, no. 55 in Forschung und wissenschaftliches Rechnen - GWDG Berichte. Gesellschaft für wissenschaftliche Datenverarbeitung mbH, Göttingen, 2000. in press.

**Conference Publications**

3. NEUMANN, D., HALTNER, E., LEHR, C.-M., KOHLBACHER, O., AND LENHOF, H.-P. Investigating the sugar-lectin interaction by computational chemistry: Tunneling the epithelial barrier. In *Abstracts of the 18th Interlec Meeting* (1998), p. 549.
4. BOGHOSSIAN, N., KOHLBACHER, O., AND LENHOF, H.-P. BALL: Biochemical Algorithms Library. In *Algorithm Engineering, 3rd International Workshop, WAE'99, Proceedings* (1999), J. Vitter and C. Zaroliagis, Eds., vol. 1668 of *Lecture Notes in Computer Science (LNCS)*, Springer, pp. 330–344.
5. KOHLBACHER, O., AND LENHOF, H.-P. Rapid software prototyping in computational molecular biology. In *Proceedings of the German Conference on Bioinformatics (GCB'99)* (1999).
6. ALTHAUS, E., KOHLBACHER, O., LENHOF, H.-P., AND MÜLLER, P. A combinatorial approach to protein docking with flexible side-chains. In *RECOMB 2000 – Proceedings of the Fourth Annual International Conference on Computational Molecular Biology* (2000), R. Shamir, S. Miyano, S. Istrail, P. Pevzner, and M. Waterman, Eds., ACM press, pp. 15–24.
7. NEUMANN, D., KOHLBACHER, O., HALTNER, E., LENHOF, H.-P., AND LEHR, C.-M. Modeling the sugar lectin interaction by computational chemistry relevant to drug design. In *Proc. 3rd World Meeting on Pharmaceutics, Biopharmaceutics and Pharmaceutical Technology* (Apr 2000), p. 233.

8. KOHLBACHER, O., BURCHARDT, A., MOLL, A., HILDEBRANDT, A., BAYER, P., AND LENHOF, H.-P. A NMR-spectra-based scoring function for protein docking. In *RECOMB 2001 – Proceedings of the Fifth Annual International Conference on Computational Molecular Biology* (2001), D. Sankoff and T. Lengauer, Eds., ACM press. (in press).

**Technical Reports**

9. BOGHOSSIAN, N., KOHLBACHER, O., AND LENHOF, H.-P. BALL: Biochemical Algorithms Library. Tech. Rep. MPI-I-99-1-002, Max-Planck-Institut für Informatik, Saarbrücken, 1999.
10. ALTHAUS, E., KOHLBACHER, O., LENHOF, H.-P., AND MÜLLER, P. A branch-and-cut algorithm for the optimal solution of the side-chain placement problem. Tech. Rep. MPI-I-2000-1-001, Max-Planck-Institut für Informatik, Saarbrücken, 2000.

**Work in Progress**

11. ALTHAUS, E., KOHLBACHER, O., LENHOF, H.-P., AND MÜLLER, P. A combinatorial approach to protein docking with flexible side-chains. submitted to J. Comput. Biol, 2000.
12. BOGHOSSIAN, N., KOHLBACHER, O., AND LENHOF, H.-P. Rapid Software Prototyping in Molecular Modeling using the Biochemical Algorithms Library (BALL). submitted to J. Exptl. Algorithmics, 2000.
13. KOHLBACHER, O., BURCHARDT, A., MOLL, A., HILDEBRANDT, A., BAYER, P., AND LENHOF, H.-P. Structure prediction of protein complexes by a NMR-based protein docking algorithm. submitted to J. Biomol. NMR, 2000.
14. NEUMANN, D., KOHLBACHER, O., LENHOF, H.-P., AND LEHR, C.-M. Protein-sugar interactions: Calculated versus experimental binding energies. submitted to J. Biol. Chem., 2000.

**Theses**

15. KOHLBACHER, O. ab-initio-Rechnungen an Silikat-Clustern: Untersuchungen zur Simulation der Adsorption an $SiO_2$. Universität des Saarlandes. Diplomarbeit, advisor: Prof. Dr. H. D. Breuer, Jan 1996.

# Bibliography


[1] R. Abagyan and M. Totrov. Biased probability Monte Carlo conformational searches and electrostatic calculations for peptides and proteins. *J. Mol. Biol.*, 235:983–1002, 1994.

[2] E. Althaus, O. Kohlbacher, H.-P. Lenhof, and P. Müller. A branch and cut algorithm for the optimal solution of the side–chain placement problem. Technical Report MPI-I-2000-1-001, Max-Planck-Institut für Informatik, Saarbrücken, 2000.

[3] E. Althaus, O. Kohlbacher, H.-P. Lenhof, and P. Müller. A combinatorial approach to protein docking with flexible side–chains. In R. Shamir, S. Miyano, S. Istrail, P. Pevzner, and M. Waterman, editors, *RECOMB 2000 – Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, pages 15–24. ACM press, 2000.

[4] E. Althaus, O. Kohlbacher, H.-P. Lenhof, and P. Müller. A combinatorial approach to protein docking with flexible side–chains. *J. Comput. Biol.*, 2000. (submitted).

[5] Programming Languages – C++. International Standard, American National Standards Institute, New York, July 1998. Ref. No. ISO/IEC 14882:1998(E).

[6] H. Balzert. *Lehrbuch der Software-Technik*, volume 2. Spektrum Akakemischer Verlag, 1996.

[7] P. Bayer. personal communication, 2000.

[8] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 129–140. Usenix, 1996.

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

[10] R. E. Berry and B. A. E. Meekings. A style analysis of C programs. *Comm. ACM*, 28(1):80–88, 1985.

[11] M. J. Betts and M. J. E. Sternberg. An analysis of conformational changes on protein-protein association: implications for predictive docking. *Protein Engineering*, 12(4):271–283, 1999.

[12] Bioperl. `http://bioperl.org`.

[13] Biopython. `http://biopython.org`.

[14] F. Bloch, W. W. Hansen, and M. E. Packard. Nuclear induction. *Phys. Rev.*, 69:127, 1946.

[15] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *J. Comput. Chem.*, 4(2):187–217, 1983.

[16] R. E. Bruccoleri and J. Novotny. Antibody modeling using the conformational search program CONGEN. *Immunomethods*, 1:96–106, 1992.

[17] A. D. Buckingham. Chemical shifts in the nuclear magnetic resonance spectra of molecules containing polar groups. *Can. J. Chem.*, 38:300–307, 1960.

[18] W. Chang, I. Shindyalov, C. Pu, and P. Bourne. Design and application of PDBLib, a C++ macromolecular class library. *CABIOS*, 10(6):575–586, 1994.

[19] S. Chidamber and C. Kemerer. Towards a metrics suite for object oriented design. *SIGPLAN Notices*, 26(11):197–211, 1991.

[20] M. L. Connolly. Shape complementarity at the hemoglobin $\alpha_1\beta_1$ subunit interface. *Biopolymers*, 25:1229–1247, 1986.

[21] J. O. Coplien. *Advanced C++ programming styles and idioms*. Addison-Wesley, 1992.

[22] W. D. Cornell, P. Cieplak, C. I. Bayly, I. R. Gould, K. M. Merz Jr., D. M. Ferguson, D. C. Spellmeyer, T. Fox, J. W. Caldwell, and P. A. Kollman. A second generation force field for the simulation of proteins, nucleic acids and organic molecules. *J. Am. Chem. Soc.*, 117:5179–5197, 1995.

[23] B. Coulange. *Software reuse*. Springer, London, 1997.

[24] The concurrent versions system. `http://www.sourcegear.com/CVS`.

[25] J. Desmet, M. D. Maeyer, B. Hazes, and I. Lasters. The dead-end elimination theorem and its use in the protein side-chain positioning. *Nature*, 356:539–542, April 1992.

[26] N. Drakos. The LaTeX to HTML translator. Technical report, Computer Based Learning Unit, University of Leeds, Jan. 1994.

[27] P. Dubois. CXX version 4.2. `http://CXX.sourceforge.net/`, 2000.

[28] R. L. Dunbrack and F. E. Cohen. Bayesian statistical analysis of protein side-chain rotamer preferences. *Protein Science*, 6:1661–1681, 1997.

[29] P. Ehrlich. On immunity with special reference to cell life. *Proceedings of the Royal Society of London*, 66:424–48, 1900.

[30] B. Elshorst, M. Hennig, H. Foersterling, A. Diener, M. Maurer, S. P., H. Schwalbe, C. Griesinger, J. Krebs, H. Schmid, T. Vorherr, and E. Carafoli. NMR solution structure of a complex of calmodulin with a binding peptide of the $Ca^{2+}$-pump. *Biochemistry*, 38:12320–12332, 1999.

[31] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. Technical Report MPI-I-98-1-007, Max-Planck-Institut für Informatik, Saarbrücken, Feb. 1998.

[32] D. Fischer, S. L. Lin, H. J. Wolfson, and R. Nussinov. A geometry-based suite of molecular docking processes. *J. Mol. Biol.*, 248:459–477, 1995.

[33] D. Fischer, R. Norel, R. Nussinov, and H. J. Wolfson. 3-D docking of protein molecules. In *Lecture Notes in Computer Science 684*, pages 20–34. Springer Verlag, New York, 1993.

[34] E. Fischer. Einfluss der Konfiguration auf die Wirkung der Enzyme. *Berichte der deutschen chemische Gesellschaft*, 27:2985–2993, 1894.

[35] R. Fleischmann, M. Adams, O. White, E. Kirkness, A. Kerlavage, C. Bult, J. Tomb, B. Dougherty, and J. Merrick. Whole-genome random sequencing and assembly of Haemophilus influenzae Rd. *Science*, 269(5223):496–512, 1995.

[36] M. Fowler and K. Scott. *UML distilled*. Addison Wesley, 1997.

[37] Free Software Foundation. GNU autoconf. `http://www.gnu.org/software/autoconf/autoconf.html`, 2000.

[38] D. Frenkel and B. Smit. *Understanding Molecular Simulations*. Academic Press, San Diego, CA, 1996.

[39] H. Friebolin. *Basic one- and two-dimensional NMR spectroscopy*. VCH, Weinheim, 1993.

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

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[42] M. Goldman. *Quantum Description of High-Resolution NMR in Liquids*. Oxford University Press, London, 1988.

[43] H. Goldstein. *Classical Mechanics*. Addison Wesley, 2nd edition, 1980.

[44] M. Gradwell and J. Feeney. Validation of the use of intermolecular NOE constraints for obtaining docked structures of protein-ligand complexes. *J. Biomol. NMR*, 7(1):48–58, 1996.

[45] H. Günther. *NMR-Spektroskopie*. Thieme, Stuttgart, 1992.

[46] C. W. Haigh and R. B. Mallion. New tables of ring current shielding in proton magnetic resonance. *Org. Mag. Res.*, 4(2):203–228, 1972.

[47] C. W. Haigh and R. B. Mallion. Ring current theories in nuclear magnetic resonance. *Prog. NMR. Spec.*, 13:303–344, 1980.

[48] R. K. Harris. *Nuclear Magnetic Resonance Spectroscopy*. Longman, London, 1994.

[49] W. Harrison and C. Cook. A note on the Berry-Meekings style metric. *Comm. ACM*, 29(2):123–125, 1986.

[50] L. Holm and C. Sander. Fast and simple monte carlo algorithm for side-chain optimization in proteins: application to model building by homology. *Proteins*, 14:213–223, 1992.

[51] W. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *J. Molecular Graphics*, 14:33–38, 1996.

[52] ILOG. *ILOG CPLEX 6.5 : user's manual*. ILOG, Bad Homburg, march 1999 edition, 1999.

[53] R. M. Jackson, H. A. Gabb, and M. J. E. Sternberg. Rapid refinement of protein interfaces incorporating solvation: Application to the protein docking problem. *J. Mol. Biol.*, 276:265–285, 1998.

[54] R. M. Jackson and M. J. E. Sternberg. A continuum model for protein-protein interactions: Application to the protein docking problem. *J. Mol. Biol.*, 250:258–275, 1995.

[55] P. Jalote. Synthesizing implementations of abstract data types from axiomatic specifications. *Software Practice and Experience*, 17(11):847–858, 1987.

[56] P. Jalote. *An integrated approach to software engineering*. Undergraduate texts in computer science. Springer, 2nd ed. edition, 1997.

[57] F. Jensen. *Introduction to Computational Chemistry*. John Wiley & Sons, 1998.

[58] C. E. Johnson and F. A. Bovey. *Chem. Phys.*, 29(5):1012, 1958.

[59] M. Jünger and S. Thienel. Introduction to ABACUS - A Branch-And-CUt System. Technical report, Informatik, Universität zu Köln, 1997.

[60] E. Katchalski-Katzir, I. Shariv, M. Eisenstein, A. A. Friesem, C. Afalo, and I. A. Vakser. Molecular surface recognition: Determination of geometric fit between proteins and their ligands by correlation techniques. *Proc. Natl. Acad. Sci. USA*, 89:2195–2199, 1992.

[61] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley, Reading, MA, USA, 1993.

[62] P. Koehl and M. Delarue. Application of a self-consistent mean field theory to predict protein side-chains conformation and estimate their conformational entropy. *J. Mol. Biol*, 239:249–275, 1994.

[63] O. Kohlbacher. BALL Coding Conventions, version 1.1. `http://www.mpi-sb.mpg.de/BALL/DOWNLOAD/-coding_1.1.ps`, 2000.

[64] D. E. J. Koshland. Application of a theory of enzyme specificity to protein synthesis. *Proc. Natl. Acad. Sci. (USA)*, 44:98–104, 1958.

[65] C. A. Laughton. Prediction of protein side-chain conformations from local three-dimensional homology relationships. *J. Mol. Biol.*, 235:1088–1097, 1994.

[66] T. Lazaridis and M. Karplus. Effective energy function for proteins in solution. *Proteins*, 35(2):133–152, 1999.

[67] A. R. Leach. Ligand docking to proteins with discrete side-chain flexibility. *J. Mol. Biol.*, 235:345–356, 1994.

[68] A. R. Leach. *Molecular Modeling: Principles and Applications*. Addison Wesley Longman, Essex, 1996.

[69] A. R. Leach and A. P. Lemon. Exploring the conformational space of protein side chains using dead-end elimination and the A* algorithm. *Proteins: Struct., Function, and Genet.*, 33:227–239, 1998.

[70] H.-P. Lenhof. An algorithm for the protein docking problem. In D. Schomburg and U. Lessel, editors, *Bioinformatics: From nucleic acids and proteins to cell metabolism. GBF Monographs Volume 18*, pages 125–139, 1995.

[71] H.-P. Lenhof. New contact measures for the protein docking problem. In *Proc. of the First Annual International Conference on Computational Molecular Biology RECOMB 97*, pages 182–191, 1997.

[72] T. Littlefair. C and C++ Code Counter. `http://cccc.netpedia.net/`, version 3pre6b, 2000.

[73] R. C. Martin. The interface segregation principle. *C++ report*, (7), Aug. 1996.

[74] C. K. Mathews and K. van Holde. *Biochemistry*. The Benjamin/Cummings Publishing Company, 1990.

[75] T. J. McCabe. A complexity measure. *IEEE Transactions on Software*, 2(4):308–320, 1976.

[76] H. M. McConnell. *J. Chem. Phys.*, 27:227–229, 1957.

[77] S. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 1993.

[78] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA user manual: version 3.8*. Max-Planck-Institut für Informatik, Saarbrücken, 1999.

[79] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, New Jersey, 2nd edition, 1997.

[80] M. Meyer, P. Wilson, and D. Schomburg. Hydrogen bonding and molecular surface shape complementarity as a basis for protein docking. *J. Mol. Biol.*, 264(1):199–210, 1996.

[81] S. Meyers. *Effective C++*. Addison-Wesley, Reading, MA, 1998.

[82] X. Morelli, A. Dolla, M. Czjzek, P. N. Palma, F. Blasco, L. Krippahl, J. J. G. Moura, and F. Guerlesquin. Heteronuclear NMR and soft docking: An experimental approach for a structural model of the cytochrome $c_{553}$-ferredoxin complex. *Biochemistry*, 39:2530–2537, 2000.

[83] D. R. Musser and A. Saini. *STL tutorial and reference guide : C++ programming with the standard template library*. Addison-Wesley professional computing ser. Addison-Wesley, Reading, MA, 1996.

[84] D. R. Musser and A. A. Stepanov. Generic programming. In P. Gianni, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, volume 358 of *LNCS*, pages 13–25. Springer, 1989.

[85] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[86] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison–Wesley, Reading, MA, 1993.

[87] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, New York; Chichester; Brisbane, 1988.

[88] A. Nicholls and B. Honig. A rapid finite difference algorithm, utilizing successive over-relaxation to solve the Poisson-Boltzmann equation. *J. Comput. Chem.*, 12(4):435–445, 1991.

[89] R. Norel, S. L. Lin, D. Xu, H. J. Wolfson, and R. Nussinov. Molecular surface variability and induced conformational changes upon protein-protein association. In R. H. Sarma and M. H. Sarma, editors, *Structure, Motion, Interaction and Expression of Biological Macromolecules. Proceedings of the Tenth Conversation. State University of New York*, pages 33–51. Adenine Press, 1998.

[90] Object Management Group. Unified modeling language specification, 1998. `http://www.omg.org`.

[91] P. W. Oman and C. R. Cook. A programming style taxonomy. *Journal of Systems and Software*, 15(3):287–301, 1991.

[92] K. Ösapay and D. Case. Analysis of proton chemical shifts in regular secondary structure of proteins. *J. Biomol. NMR*, 4:215–230, 1994.

[93] K. Ösapay and D. A. Case. A new analysis of proton chemical shifts in proteins. *J. Am. Chem. Soc.*, 113:9436–9444, 1991.

[94] M. Osawa, H. Tokumitsu, M. B. Swindells, H. Kurihara, M. Orita, T. Shibanuma, T. Furuya, and M. Ikura. A novel calmodulin target recognition revealed by its NMR structure in complex with a peptide derived from $Ca^{2+}$-calmodulin-dependent protein kinase kinase. *Nat. Struct. Biol.*, 6:819, 1999.

[95] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley Publishing Company, 1994.

[96] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3), 1998.

[97] L. Pauling. *Chem. Phys.*, 4:673, 1936.

[98] V. I. Polshakov, W. D. Morgan, B. Birdsall, and J. Feeney. Validation of a new restraint docking method for solution structure determination of protein-ligand complexes. *J. Biomol. NMR*, 14:115–122, 1999.

[99] J. Ponder and F. Richards. Tertiary templates for proteins – use of packing criteria in the enumeration of allowed sequences for different structural classes. *J. Mol. Biol.*, 193:775–791, 1987.

[100] L. Prechelt. An empirical comparison of C, C++, Java, Perl, Python, rexx, and Tcl for a search/string-processing program. Technical Report 2000-5, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, 2000.

[101] E. M. Purcell, H. C. Torrey, and R. V. Pound. Resonance absorption by nuclear magnetic moments in a solid. *Phys. Rev.*, 69:37–38, 1946.

[102] QT release 2.02. `http://www.troll.no/products/qt.html`.

[103] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual*. Addison Wesley, 1999.

[104] R. R. Rustandi, A. C. Drohat, D. M. Baldisseri, P. T. Wilder, and D. J. Weber. The $Ca^{2+}$-dependent interaction of S100B($\beta\beta$) with a peptide derived from p53. *Biochemistry*, 37:1951–1960, 1998.

[105] B. Sandak, R. Nussinov, and H. J. Wolfson. A Method for Biomolecular Structural Recognition and Docking Allowing Conformational Flexibility. *J. Comp. Biol.*, 5(4):631–654, 1998.

[106] B. Sandak, H. J. Wolfson, and R. Nussinov. Flexible Docking Allowing Induced Fit in Proteins: Insights From an Open to Closed Conformational Isomers. *Proteins*, 32:159–174, 1998.

[107] R. Sayle and E. Milner-White. RASMOL: biomolecular graphics for all. *Trends Biochem. Sci.*, 20(9):374, 1995.

[108] B. R. Seavey, E. A. Farr, W. M. Westler, and J. Markley. A relational database for sequence-specific protein NMR data. *J. Biomol. NMR*, 1:217–236, 1991.

[109] D. Sitkoff, K. A. Sharp, and B. Honig. Accurate calculation of hydration free energies using macroscopic solvent models. *J. Phys. Chem.*, 98(7):1978–1988, 1994.

[110] E. D. Solometsev. Elliptic integral. In M. Hazewinkel, editor, *Encyclopedia of Mathematics*, volume 3, pages 372–373. Kluwer Academic Publishers, Dordrecht, 1987.

[111] R. Srinivasan. *XDR: External Data Representation Standard*. Internet Engineering Task Force, 1995. Request for Comments 1832.

[112] F. Stajano. Implementing the SMS server, or why I switched from tcl to python. In *Proceedings of the 7th International Python Conference*, Nov. 1998.

[113] M. J. E. Sternberg, H. A. Gabb, and R. M. Jackson. Predictive docking of protein-protein and protein-DNA complexes. *Curr. Opin. Struct. Biol.*, 8:250–256, 1998.

[114] L. Stryer. *Biochemie*. Spektrum Akademischer Verlag, 1991.

[115] SUN Microsystems. *XDR: External Data Representation Standard*. Internet Engineering Task Force, Networking Working Group, 1987. Request for Comments 1014.

[116] P. Thompson. Sip version 0.12. `http://www.thekompany.com/projects/pykde/`, 2000.

[117] M. Totrov and R. Abagyan. Detailed ab initio prediction of lysozyme-antibody complex with 1.6 Å accuracy. *Nat. Struct. Biol.*, 1:259–263, 1994.

[118] W. Vahrson, K. Hermann, J. Kleffe, and B. Wittig. Object-oriented sequence analysis: SCL – a C++ class library. *CABIOS*, 12(2):119–127, 1996.

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

[120] F. J. M. van de Ven. *Multidimensional NMR in liquids: basic principles and experimental methods*. Wiley-VCH, New York, 1995.

[121] G. van Rossum. Python version 1.5.1. `http://www.python.org`.

[122] P. von Rague Schleyer, editor. *Encyclopedia of Computational Chemistry*. John Wiley & Sons, 1998.

[123] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates Inc., 2nd edition, 1996.

[124] P. Walters and M. Stahl. BABEL version 1.6. University of Arizona.

[125] H. Wang. Grid-search molecular accessible surface algorithm for solving the docking problem. *J. Comput. Chem.*, 12:746–750, 1991.

[126] J. S. Waugh and R. W. Fessenden. *J. Am. Chem. Soc.*, 79:846, 1959.

[127] D. J. Weber, A. M. Libson, A. G. Gittis, M. Lebowitz, and A. S. Mildvan. NMR docking of a substrate into the X-ray structure of the ASP-21-GLU mutant of staphylococcal nuclease. *Biochemistry*, 33(26):8017–8028, 1994.

[128] Z. Weng, S. Vajda, and C. Delisi. Prediction of protein complexes using empirical free energy functions. *Protein Science*, 5:614–626, 1996.

[129] M. P. Williamson and T. Asakura. Empirical comparisons of models for chemical-shift calculation in proteins. *J. Mag. Res. B*, 101:63–71, 1993.

[130] M. P. Williamson, T. Asakura, E. Nakamura, and M. Demura. A method for the calculation of protein $\alpha$-CH chemical shifts. *J. Biomol. NMR*, 2:83–98, 1992.

[131] L. A. Wolsey. *Integer programming*. Wiley-interscience series in discrete mathematics and optimization. Wiley & Sons, New York, 1998.

[132] C. Zhang, J. L. Cornette, and C. DeLisi. Consistency in structural energetics of protein folding and peptide recognition. *Prot. Sci.*, 6:1059–1064, 1997.

[133] M. Zöckler and R. Wunderling. DOC++ version 3.2. `http://www.zib.de/Visual/software/doc++/`.

# Index

## Symbols


$\alpha$-helix . . . . . 11
$\beta$-pleated sheet . . . . . 11
2-deoxyribose . . . . . 14
80-20 rule . . . . . 69


### A


add_hydrogens (FragmentDB method) . . . . . 119
AddBallAndStickModel (BALL class) . . . . . 98
adenine . . . . . 14
aggregation . . . . . 130
AMBER . . . . . 17, 88
AmberBend (BALL class) . . . . . 89
AmberFF (BALL class) . . . . . 89
AmberNonBonded (BALL class) . . . . . 89
AmberStretch (BALL class) . . . . . 89
AmberTorsion (BALL class) . . . . . 89
amino acids . . . . . 10
angular momentum quantum number
    definition . . . . . 45
ANSI C++ . . . . . 70
API . . . . . 102
apply<T> (Composite method) . . . . . 79
ar . . . . . 117
association . . . . . 129
Atom (BALL class) . . . . . 67, 76, 84
AtomContainer (BALL class) . . . . . 85, 95, 96
AtomIterator (BALL class) . . . . . 85
atoms . . . . . 9
atoms (PyBALL function) . . . . . 103
AtomVector (BALL class) . . . . . 88
attributes . . . . . 129
awk . . . . . 82


### B


backbone . . . . . 10
BALL . . . . . 65–124
    AmberFF
        setup . . . . . 122
    CenterOfMassProcessor
        finish . . . . . 80
        operator () . . . . . 80
    classes
        AddBallAndStickModel . . . . . 98
        AmberBend . . . . . 89
        AmberFF . . . . . 89
        AmberNonBonded . . . . . 89
        AmberStretch . . . . . 89
        AmberTorsion . . . . . 89
        Atom . . . . . 67, 76, 84
        AtomContainer . . . . . 85, 95, 96
        AtomIterator . . . . . 85
AtomVector . . . . . 88
CanonicalMD . . . . . 93, 94
CenterOfMassProcessor . . . . . 79, 80
Chain . . . . . 85
Client . . . . . 98
Composite . . . . . 72, 73, 75, 85
composite . . . . . 72
Control . . . . . 98
CreateSpectrumProcessor . . . . . 96
DisplayProperties . . . . . 99, 100
DNA . . . . . 67, 85
Embeddable . . . . . 104, 105
EnergyMinimizer . . . . . 93
Expression . . . . . 86
ExpressionPredicate . . . . . 86
FDPB . . . . . 81
ForceField . . . . . 88, 89
ForceFieldComponent . . . . . 88, 89
Fragment . . . . . 67, 84
GeneralException . . . . . 82
HINFile . . . . . 87
INIFile . . . . . 90
INIFile . . . . . 90, 95
LennardJones . . . . . 91
LineSearch . . . . . 93
LogStream . . . . . 81
MainControl . . . . . 97, 99
MicroCanonicalMD . . . . . 93, 94
ModularWidget . . . . . 97, 98
MOL2File . . . . . 87
MolecularControl . . . . . 99, 100
MolecularDynamics . . . . . 94
MolecularDynamicsSimulation . . 93, 95
MolecularProperties . . . . . 100
MolecularStructureFile . . . . . 87
Molecule . . . . . 67, 78, 84
MyClass . . . . . 105
NMRSpectrum . . . . . 96
NucleicAcid . . . . . 85
Nucleotide . . . . . 85
Object . . . . . 74, 112
Objects . . . . . 112
OpenPDBFile . . . . . 99
OpenPDBFileDialog . . . . . 99
Options . . . . . 80, 81, 93
Parameters . . . . . 90, 91, 95
ParameterSection . . . . . 90, 91
PDBFile . . . . . 87
PersistenceManager . . . . . 74–77
PersistentObject . . . . . 74–76
Protein . . . . . 67, 78, 85
RegularExpression . . . . . 82
Residue . . . . . 85
ResourceFile . . . . . 87
RNA . . . . . 67, 85

Scene ..... 97–100
SecondaryStructure ..... 85
Selectable ..... 72, 75, 76
Selector ..... 86
Server ..... 98
ShiftModel ..... 95
ShiftModule ..... 95
SnapShotManager ..... 95
SocketStream ..... 78
String ..... 82
Substring ..... 82
System ..... 67, 84, 85, 87, 114
TextPersistenceManager ..... 75–77
TimeStamp ..... 73
UnaryProcessor<Atom> ..... 79, 80, 86
UnaryProcessor<T> ..... 79
XDRPersistenceManager ..... 75, 77
XYZFile ..... 87
Composite
apply<T> ..... 79
contains_selection_ ..... 72, 73
containsSelection ..... 72
Embeddable
countInstances ..... 105
getInstance ..... 105
registerThis ..... 104, 105
unregisterThis ..... 104
exceptions
DivisionByZero ..... 114
FDPB
options ..... 81
files
classTest.h ..... 113
COMMON/global.h ..... 71
config.h ..... 71
ForceField
options ..... 88
setup ..... 88, 89
updateEnergy ..... 88, 89
updateForces ..... 88, 89
FragmentDB
add_hydrogens ..... 119
functions
isInstanceOf ..... 112
isInstanceOf<T> ..... 78
isKindOf ..... 112
isKindOf<T> ..... 78
LogStream
info ..... 82
setPrefix ..... 82
macros
BALL_EMBEDDABLE ..... 104
CHECK ..... 113, 114
END_TEST ..... 113, 114
RESULT ..... 113, 114
START_TEST ..... 113, 114
TEST_EQUAL ..... 113
TEST_EXCEPTION ..... 114
TEST_FILE ..... 114
TEST_NOT_EQUAL ..... 114
TEST_REAL_EQUAL ..... 114
ModularWidget
initializeWidget ..... 97
namespaces
Constants ..... 71
RTTI ..... 78
Object
global_handle_ ..... 74
handle_ ..... 74
PersistenceManager
get ..... 76
put ..... 76
PersistentObject
finalize ..... 74
persistentRead ..... 74, 76
persistentWrite ..... 74, 76
read ..... 76
write ..... 76
Selectable
deselect ..... 72
select ..... 72
System
beginAtom ..... 85
endAtom ..... 85
TimeStamp
stamp ..... 73
types
PointerSizeInt ..... 71
UnaryOperator
operator () ..... 86
UnaryProcessor¡T¿
finish ..... 79
operator () ..... 79
start ..... 79
BALL_EMBEDDABLE (BALL macro) ..... 104
BALLVIEW
definition ..... 96
base ..... 14
base pair ..... 14
basic components ..... 70
beginAtom (System method) ..... 85
bend energy ..... 17
big endian ..... 77
binding site ..... 28
biomolecules ..... 9
BioPerl ..... 5, 101
BioPython ..... 5, 101
black box testing ..... 111
bonded interactions ..... 14
bovine pancreatic trypsin inhibitor ..... 3
BPTI ..... 3
branch-&-cut algorithm ..... 29
builder ..... 74, 75


**C**


C-terminus ..... 10
canonical ensemble
definition ..... 94
CanonicalMD (BALL class) ..... 93, 94
CCCC ..... 107
CenterOfMassProcessor (BALL class) ..... 79, 80
cerr ..... 81
Chain (BALL class) ..... 85

chain placement problem . . . 26
CHARMM . . . 88
CHECK (BALL macro) . . . 113, 114
chemical bond . . . 9
chemical shift . . . 24
    definition . . . 48
class . . . 129
class library . . . 67
classTest.h (BALL file) . . . 113
Client (BALL class) . . . 98
collaboration . . . 130
COM
    definition . . . 108
comment lines
    definition . . . 108
COMMON/global.h (BALL file) . . . 71
Composite (BALL class) . . . 72, 73, 75, 85
composite (BALL class) . . . 72
composite design pattern . . . 72
composition . . . 130
concrete builder . . . 75
concrete creator . . . 75
concrete product . . . 75
Concurrent Revision System . . . 107
config.h (BALL file) . . . 71
configure . . . 115, 117
conjugate gradient . . . 92
Constants (BALL namespace) . . . 71
constraints . . . 31
contains_selection_ (Composite attribute) 72, 73
containsSelection (Composite method) . . . 72
continuum model . . . 38
Control (BALL class) . . . 98
Coulomb's law . . . 16
countInstances (Embeddable method) . . . 105
cout . . . 81
CreateSpectrumProcessor (BALL class) . . . 96
creator . . . 75
cutting plane . . . 32
CVS . . . 107, 115
CWEB . . . 108
CXX . . . 103
cyclomatic complexity . . . 108
cytosine . . . 14

## D

dead-end elimination theorem . . . 29
deadly diamond
    definition . . . 73
DEE . . . 29
Democritus of Abdera . . . 9
deoxyribonucleic acid . . . 13
deoxyribose . . . 14
dependency . . . 130
deselect (Selectable method) . . . 72
design pattern
    composite . . . 72
design patterns . . . 130
    builder . . . 74
    definition . . . 68
    factory method . . . 74
diffusion coefficients . . . 94
director . . . 75
DisplayProperties (BALL class) . . . 99, 100
DivisionByZero (BALL class) . . . 114
DNA . . . 13
DNA (BALL class) . . . 67, 85
DOC++ . . . 108
Docking
    protein-protein . . . 25
    semi-flexible . . . 25
docking . . . 19–61, 64
    flexible . . . 25–45
double helix . . . 14
drug design . . . 2
dynamic_cast . . . 78

## E

edit . . . 120
effective field . . . 47
electron microscopy . . . 45
electrostatic energy . . . 17
Embeddable (BALL class) . . . 104, 105
Embeddable (PyBALL class) . . . 105
Embedding . . . 104
Emil Fischer . . . 2
END_TEST (BALL macro) . . . 113, 114
endAtom (System method) . . . 85
energy generator . . . 88
Energy minimization . . . 92
EnergyMinimizer (BALL class) . . . 93
ensemble . . . 94
ethanol . . . 9
exhaustive search . . . 26
Expression (BALL class) . . . 86
ExpressionPredicate (BALL class) . . . 86
extension modules . . . 102
External Data Representation . . . 77
extractors
    definition . . . 103

## F

factory method . . . 74
false positives
    definition . . . 21
FAQ list . . . 109
FDPB (BALL class) . . . 81
field sweep . . . 48
finalize (PersistentObject method) . . . 74
finish (CenterOfMassProcessor method) . . . 80
finish (UnaryProcessor<T> attribute) . . . 79
finish (UnaryProcessor<T> method) . . . 79
for loop (Python) . . . 102
force . . . 16
force field . . . 16, 17
force generator . . . 88
ForceField (BALL class) . . . 88, 89
ForceFieldComponent (BALL class) . . . 88, 89
Foundation Classes . . . 70
Fourier transform . . . 45
Fragment (BALL class) . . . 67, 84

framework . . . . . 67
frequency sweep . . . . . 49
functional testing . . . . . 111

**G**

`GeneralException` (BALL class) . . . . . 82
generalization . . . . . 130
generic programming
definition . . . . . 68
genomics . . . . . 2
geometry optimization . . . . . 92
`get` (`PersistenceManager` method) . . . . . 76
`getInstance` (`Embeddable` method) . . . . . 105
global minimum energy conformation . . . . . 26
`global_handle_` (`Object` attribute) . . . . . 74
GMEC . . . . . 26
GMEC polyhedron
definition . . . . . 32
GNU autoconf . . . . . 117
graphical user interface . . . . . 96
guanine . . . . . 14
GUI . . . . . 96
Guido van Rossum . . . . . 102
gyromagnetic ratio . . . . . 46

**H**

Haemophilus influenzae . . . . . 2
`handle_` (`Object` attribute) . . . . . 74
`HINFile` (BALL class) . . . . . 87
human genome . . . . . 2
hydroxyl group . . . . . 9

**I**

ILP . . . . . 29
immutable . . . . . 103
induced fit . . . . . 19, 21
`info` (`LogStream` method) . . . . . 82
`INIFile` (BALL class) . . . . . 90
`INIFile` (BALL class) . . . . . 90, 95
`initializeWidget` (`ModularWidget` method) . . 97
integer linear program . . . . . 31
Integer Linear Programming . . . . . 29
interface pollution . . . . . 79
intermolecular exchange . . . . . 53
intramolecular interactions . . . . . 14
ions
definition . . . . . 15
`isInstanceOf` (BALL function) . . . . . 112
`isInstanceOf<T>` (BALL function) . . . . . 78
`isKindOf` (BALL function) . . . . . 112
`isKindOf<T>` (BALL function) . . . . . 78
isolated system . . . . . 94
iterator traits . . . . . 79
iterators . . . . . 79

**K**

kernel
iterators . . . . . 85
kernel classes . . . . . 70

**L**

L_C
definition . . . . . 108
Larmor frequency . . . . . 47
Larmor precession
definition . . . . . 47
`LennardJones` (BALL class) . . . . . 91
Leucippus of Miletus . . . . . 9
Life Science . . . . . 2
lines of code
definition . . . . . 108
`LineSearch` (BALL class) . . . . . 93
`link` . . . . . 119
little endian . . . . . 77
LOC
definition . . . . . 108
lock-and-key principle . . . . . 2
definition . . . . . 19
`LogStream` (BALL class) . . . . . 81
Lorentzian . . . . . 53
Lotus Notes . . . . . 109

**M**

magnetic moment . . . . . 46
magnetic quantum number
definition . . . . . 46
`MainControl` (BALL class) . . . . . 97, 99
methyl group . . . . . 9
microcanonical ensemble
definition . . . . . 94
`MicroCanonicalMD` (BALL class) . . . . . 93, 94
MMTK . . . . . 5
`ModularWidget` (BALL class) . . . . . 97, 98
`MOL2File` (BALL class) . . . . . 87
Molecular Dynamics Simulation . . . . . 94
Molecular Modeling Toolkit . . . . . 5
molecular surface . . . . . 38
`MolecularControl` (BALL class) . . . . . 99, 100
`MolecularDynamics` (BALL class) . . . . . 94
`MolecularDynamicsSimulation` (BALL class) . 93, 95
`MolecularProperties` (BALL class) . . . . . 100
`MolecularStructureFile` (BALL class) . . . . . 87
`Molecule` (BALL class) . . . . . 67, 78, 84
molecule . . . . . 9
MOLVIEW . . . . . 96
molview
definition . . . . . 96
`molview` . . . . . 78, 81, 98, 100, 101, 104
monomers . . . . . 10
multi-greedy . . . . . 29
multiple inheritance . . . . . 73
`MyClass` (BALL class) . . . . . 105

**N**

N-terminus . . . . . 10
native structure

definition . . . 3
navigable . . . 130
Newton-Raphson . . . 92
NMR . . . 24
NMR spectra . . . 24
NMRSpectrum (BALL class) . . . 96
NOE . . . 24
nonbonded interactions . . . 14
nuclear angular momentum . . . 45
Nuclear Magnetic Resonance . . . 24, 45
nuclear Overhauser effect . . . 24
nucleic acids . . . 9, 13
NucleicAcid (BALL class) . . . 85
Nucleotide (BALL class) . . . 85
nucleotide . . . 13
nuclide . . . 45

**O**

Object (BALL class) . . . 74, 112
Objects (BALL class) . . . 112
one-letter code . . . 11
OpenPDBFile (BALL class) . . . 99
OpenPDBFileDialog (BALL class) . . . 99
operations . . . 129
operator () (CenterOfMassProcessor method) 80
operator () (UnaryOperator method) . . . 86
operator () (UnaryProcessor<T> method) . . . 79
Options (BALL class) . . . 80, 81, 93
options (FDPB attribute) . . . 81
options (ForceField attribute) . . . 88
orthodox canonical class interface . . . 107
ostream . . . 74
ostream . . . 81

**P**

packages
definition . . . 130
Parameters (BALL class) . . . 90, 91, 95
ParameterSection (BALL class) . . . 90, 91
parm . . . 121
partial charges . . . 15
pauli exclusion principle . . . 16
PDBFile (BALL class) . . . 87
PDBLib . . . 5
peptide bond . . . 10
Perl . . . 101
persistence . . . 73
PersistenceManager (BALL class) . . . 74–77
PersistentObject (BALL class) . . . 74–76
persistentRead (PersistentObject method) 74, 76
persistentWrite (PersistentObject method) 74, 76
phosphate . . . 14
PointerSizeInt (BALL type) . . . 71
Poisson-Boltzmann equation . . . 39
polyhedron
definition . . . 32
precession . . . 47
predicates . . . 86
primary structure . . . 10
primitive tests . . . 112
processor . . . 79
product . . . 75
Protein (BALL class) . . . 67, 78, 85
protein . . . 10
protein-protein docking problem
definition . . . 2
proteins . . . 9
proteomics
definition . . . 2
protonate . . . 119
protons . . . 46
put (PersistenceManager method) . . . 76
PyBALL
classes
Embeddable . . . 105
functions
atoms . . . 103
Python . . . 101
for loop . . . 102
functions
range . . . 102
types
tuple . . . 103

**Q**

quantum number . . . 45
quarternary structure . . . 11
Quasi-Newton . . . 92

**R**

RAD . . . 67
radial distribution functions . . . 94
Ramachandran plot . . . 26
random coil shift . . . 51
range (Python function) . . . 102
Rapid Application Development . . . 67
Rapid Software Prototyping . . . 67
RasMol . . . 96
read (PersistentObject method) . . . 76
realization . . . 130
receiver coils . . . 48
reference manual . . . 109
registerThis (Embeddable method) . . . 104, 105
regular expression . . . 114
RegularExpression (BALL class) . . . 82
repository . . . 115
Residue (BALL class) . . . 85
residues . . . 10
resonance frequency . . . 46
ResourceFile (BALL class) . . . 87
RESULT (BALL macro) . . . 113, 114
ribonucleic acid . . . 13
ribose . . . 14
rigid docking . . . 2
rigid-body docking
definition . . . 19
RMSD . . . 21

RNA . . . . . 13, 14
RNA (BALL class) . . . . . 67, 85
rotamer graphs . . . . . 31
rotamer library . . . . . 26
rotamers . . . . . 26
RPC . . . . . 77
RTTI (BALL namespace) . . . . . 78

## S

sander . . . . . 121
Scene (BALL class) . . . . . 97–100
SCL . . . . . 5
scripting languages . . . . . 101
SDK . . . . . 5
secondary structures . . . . . 11
SecondaryStructure (BALL class) . . . . . 85
sed . . . . . 117
select (Selectable method) . . . . . 72
Selectable (BALL class) . . . . . 72, 75, 76
selection . . . . . 72, 93
Selector (BALL class) . . . . . 86
semi-flexible docking
    definition . . . . . 3, 25
sequence . . . . . 10
sequence object . . . . . 102
serialization . . . . . 73
Server (BALL class) . . . . . 98
setPrefix (LogStream method) . . . . . 82
setup (AmberFF method) . . . . . 122
setup (ForceField method) . . . . . 88, 89
sh . . . . . 117
shielding constant . . . . . 47
shift assignment . . . . . 49
    definition . . . . . 24
ShiftModel (BALL class) . . . . . 95
ShiftModule (BALL class) . . . . . 95
side chain . . . . . 10
side chain demangling . . . . . 25
side chain flexibility . . . . . 25
signal-slot mechanism . . . . . 98
SIP . . . . . 103
SnapShotManager (BALL class) . . . . . 95
SocketStream (BALL class) . . . . . 78
software metrics . . . . . 107
solvent excluded surface . . . . . 38
spin quantum number
    definition . . . . . 45
stamp (TimeStamp method) . . . . . 73
standard predicates . . . . . 86
Standard Template Library . . . . . 70
start (UnaryProcessor<T> method) . . . . . 79
START_TEST (BALL macro) . . . . . 113, 114
static class attributes . . . . . 73
steepest descent . . . . . 92
STL . . . . . 70
    iterators . . . . . 85
stretch energy . . . . . 17
String (BALL class) . . . . . 82
structural testing . . . . . 111
style metrics . . . . . 107
Substring (BALL class) . . . . . 82
subtest . . . . . 112
sugar . . . . . 14
sweep coils . . . . . 48
SWIG . . . . . 103
System (BALL class) . . . . . 67, 84, 85, 87, 114

## T

Tcl . . . . . 101
tertiary structure . . . . . 11
test cases . . . . . 111
test oracle . . . . . 111
TEST_EQUAL (BALL macro) . . . . . 113
TEST_EXCEPTION (BALL macro) . . . . . 114
TEST_FILE (BALL macro) . . . . . 114
TEST_NOT_EQUAL (BALL macro) . . . . . 114
TEST_REAL_EQUAL (BALL macro) . . . . . 114
testing
    black box . . . . . 111
    functional . . . . . 111
    structural . . . . . 111
    white box . . . . . 111
TextPersistenceManager (BALL class) . . . . . 75–77
Three Line Rule
    definition . . . . . 68
three-letter code . . . . . 11
thymine . . . . . 14
time stamps . . . . . 73
TimeStamp (BALL class) . . . . . 73
torsion angle . . . . . 17
torsion energy . . . . . 17
torsional flexibility . . . . . 26
torsions . . . . . 26
trajectory . . . . . 94
transmitter coil . . . . . 48
true positives
    definition . . . . . 21
trypsin . . . . . 3
tuple (Python type) . . . . . 103
tuples . . . . . 102
tutorial . . . . . 109
typeid . . . . . 78

## U

UML . . . . . 129
UnaryProcessor<Atom> (BALL class) . . . . 79, 80, 86
UnaryProcessor<T> (BALL class) . . . . . 79
Unified Modeling Language . . . . . 129
unregisterThis (Embeddable method) . . . . . 104
updateEnergy (ForceField method) . . . . . 88, 89
updateForces (ForceField method) . . . . . 88, 89
uracil . . . . . 14

## V

van der Waals energy . . . . . 17
van der Waals interactions . . . . . 16
VIEW . . . . . 96
virtual constructor . . . . . 75
virtual inheritance . . . . . 74
visitor pattern . . . . . 79

VMD . . . . . 96

## W

weakly typed . . . . . 102
white box testing . . . . . 111
widgets . . . . . 97
wrapper classes . . . . . 103
`write` (`PersistentObject` method) . . . . . 76

## X

X-ray crystallography . . . . . 45
XDR . . . . . 77
`XDRPersistenceManager` (BALL class) . . . . . 75, 77
`XYZFile` (BALL class) . . . . . 87

## Z

zwitterion . . . . . 10