BALL—rapid software prototyping in computational molecular biology

Oliver Kohlbacher* and Hans-Peter Lenhof

Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

Received on December 3, 1999; revised on April 22, 2000; accepted on June 18, 2000

Abstract

Motivation: Rapid software prototyping can significantly reduce development times in the field of computational molecular biology and molecular modeling. Biochemical Algorithms Library (BALL) is an application framework in C++ that has been specifically designed for this purpose.

Results: 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. 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. BALL is well suited to serve as a public repository for reliable data structures and algorithms. We show in an example that the implementation of complex methods is greatly simplified when using the data structures and functionality provided by BALL.

Availability: BALL is available via internet from http://www.mpi-sb.mpg.de/BALL/. It may be used free of charge for research and teaching. Commercial licenses are available upon request.

Contact: O.Kohlbacher, oliver@mpi-sb.mpg.de

Introduction

Implementation is often the biggest hurdle that has to be overcome in order to test and verify new ideas and methods. When implementing new methods, a remarkable amount of time is spent with the implementation of basic data structures, standard methods, and ‘glue code’ to adapt user code to existing software. Tools for rapid software prototyping accelerate the implementation of new ideas. In this work, we present a C++ application framework for rapid software prototyping in the area of molecular modeling and computational molecular biology, called Biochemical Algorithms Library (BALL). A thorough analysis of the implementation process and existing software led us to the formulation of four basic design principles for BALL: ease of use, functionality, openness, and robustness.

Ease of use is crucial for the acceptance of a new tool. It should therefore employ a widely used programming language like Fortran, C, Java, or C++. The interface has to be intuitive, consistent, and well documented. An object-oriented language should be preferred, as object-orientation simplifies code reuse and results in better structured interfaces (Coulange, 1997; Meyer, 1997). Obviously functionality is of fundamental importance. Standard data structures, methods, and tools should be directly available to the user. We identified some key functionalities that are needed for many applications in computational molecular biology: file import/export, ‘molecular’ data structures, molecular mechanics, and visualization. Doubtless, no software package can provide all potentially interesting algorithms and data structures for all applications in such a diverse field as Computational Molecular Biology. Openness means that the software package should be compatible with other existing libraries. For example, it should be possible to combine it with other class libraries to take advantage of their well tested graph algorithms and geometric algorithms. But it also implies a sufficient support for the integration of external programs. For this purpose, code for reading and writing the most important file formats has to be available. Since many programs use specific file formats (e.g. for storing parameters or settings), the library is most useful, if it provides support for the parsing of field-based and column-based file formats beyond the standard I/O operations. Openness also includes extensibility, i.e. it should be simple to add new objects and functionality without changing the existing code. This goal can be reached through a clear, but nevertheless flexible, design. For example, if the user wants to add an additional force field, the library should provide the elementary functionality and a well defined interface common to all force fields, such that the user only has to fill in the force-field-specific pieces of code. The term robustness describes the code’s ability to cope with unexpected or faulty data. For example, although the format for PDB files is well specified, a multitude of

*To whom correspondence should be addressed.
variants exists. A robust PDB reader will accept these files as well and will be able to extract as much of the data as possible. Robustness also implies correctness, so a concept for software testing is also needed. Besides these main principles, efficiency and portability are also self-evident design goals. We will now briefly discuss how these requirements are met by existing software.

Out of the multitude of existing software packages we have selected four examples that illustrate the variety of approaches that can be used to implement and test new methods in molecular modeling and computational molecular biology. The first package we will discuss is the Cerius² Software Development Kit (SDK) (Molecular Simulations Inc., 1997). It provides a well defined and well documented interface for the extension of the molecular modeling environment Cerius². It provides access to the full functionality of Cerius², but it requires a license for Cerius² and is not object-oriented. The Molecular Operating Environment (MOE, The Chemical Computing Group, 2000) is a software package based on its own vector-oriented programming language (SVL). It provides functionality for sequence-related problems (searching, alignment), molecular mechanics, docking, drug design, and visualization. Its main advantage is the fact that this functionality is implemented in SVL and the user is provided with the source code to adapt existing functionality to new problems. SVL serves as scripting and extension language as well. Further extensions to SVL are possible through an API that permits the integration of C and Fortran code. Although MOE is clearly a powerful tool, its unusual programming language and the lack of object-orientation are drawbacks. PDBLib (Chang et al., 1994) is an object-oriented package that provides a rich functionality to read and manipulate PDB files. It is well suited for this purpose, but it lacks any further functionality. PDBLib is an example of the many class libraries currently available that serve just one small and very specialized purpose. A package that provides a reasonable functionality is MMTK (Hinsen, 1997). It provides an extension to the object-oriented script language Python (van Rossum, 1999) and combines the advantages of the object-oriented approach with those of an interpreted language. MMTK is easy to use, provides basic data structures, and some fundamental functionality in the fields of visualization and molecular mechanics. Due to its object-oriented concept, MMTK is open and extensible on the scripting language level. However, time-critical code sections (including most of the molecular mechanics code) are written in C, rendering them difficult to reuse. This limits the extensibility of the package. MMTK also depends on many external software packages (e.g. numerical extensions for Python, VMD for visualization).

In 1997, we started to design and implement a new object-oriented application framework for rapid software prototyping called BALL. The first decision was the choice of the programming language. We decided to prefer C++ over Java. Both languages are easy to use and object-oriented. Java is certainly the more robust language but C++ allows generic programming, operator overloading, and the resulting code naturally performs much better than similar Java code. In addition, the majority of existing libraries is written in C++, thus allowing a seamless integration of this code.

BALL differs from the software packages above in its openness, extensibility, and ease of use. We will now briefly discuss the techniques we used to address these issues.

Ease of use

Object-oriented programming greatly facilitates the design of clearly structured interfaces. Along with carefully chosen class names and identifiers, this improves the intuitive usability of BALL. Through a set of rules for choosing method names and strict adherence to certain coding styles, we obtained a very consistent interface. The use of operator overloading allows a natural and comprehensible syntax. When implementing new functionality, we always took care that frequently recurring operations could be performed with not more than three lines of code. Although this leads to interfaces that are not minimal, it significantly improves the rapid prototyping capabilities of BALL.

Another important point is the documentation. Clearly, a piece of software is worthless without documentation. The documentation of BALL is integrated into the source code and can be extracted using the tool DOC++ (Zöckler and Wunderling, 1998) as a printable document or in HTML format. The integration into the source code keeps the documentation consistent with the implementation. We also provide a tutorial for new users which explains the most important basic concepts of BALL by means of short example programs.

Openness

Since we use BALL together with several other libraries on a regular basis, we took care to ensure an unproblematic integration of external code. All BALL objects and functions are embedded in a namespace to avoid collisions with symbols of other libraries. Similarly, all preprocessor definitions are prefixed by BALL. Wherever appropriate, we used template classes and functions in order to allow the use of other data types with our code. For example, all mathematical classes are template classes. Hence, a user can decide to use the BALL vector and matrix classes together with data types of any precision he chooses (float, double, or even arbitrary precision floating point numbers from the LEDA library, Mehlhorn et al., 1998).
Robustness

Speed is always an issue in computational molecular biology. Unfortunately, robust, fault tolerant code is generally slower than comparable code without additional consistency checks. On the other hand, robust code can drastically reduce the development times and helps to uncover problems with input data that might otherwise pass unnoticed. As BALL is mainly intended for rapid software prototyping, we decided that robustness is generally more important than speed. Nevertheless, we used techniques like inlining and templates to implement this robust code as efficiently as possible. The computationally expensive parts of BALL were then hand-tuned for optimal performance. In our experience, the loss in efficiency is more than compensated by the reduced development times. We compare the speed of some algorithms in BALL to other software packages at the end of this work. BALL also provides an extensive test suite to verify the correctness of its components. We employed object-oriented software metrics to identify critical code sections and to improve the overall software quality.

Functionality

The functionality of BALL is currently focused on algorithms necessary for protein docking and drug design, but because of its extensibility, more functionality can be added (and is planned). The basis of all BALL classes is an extensive set of foundation classes. They provide generic implementations of advanced data structures (e.g. trees, hash associative containers, hash grids, etc.), mathematical objects (e.g. matrices, vectors, geometric objects), implementations of design patterns, object persistence, and access to the operating system (e.g. networking support, file handling). The BALL kernel contains the molecular data structures for the representation of atoms, bonds, molecules, proteins, etc. The kernel classes are implemented with the foundation classes and are carefully designed to provide maximum extensibility and flexibility. The third layer of classes (on top of the foundation classes and the kernel) provides the functionality required for applications. We call them basic components. Each of these basic components is independent of the other components. We have implemented five basic components that cover molecular mechanics, file import/export, advanced solvation methods, structure analysis/comparison, and visualization. These areas were selected, because our main interest stems from the field of protein docking. The molecular mechanics component does not only provide standard force fields (AMBER, CHARMM), but also a generic force field. This generic force field is a base class for all force fields. It implements fundamental methods to manage parameter files, parameter assignment, and atom type assignment, and defines a common interface for all force fields. Thus, large portions of code may be reused for the implementation of new force fields. The file import/export component implements general methods for efficient file handling, but also methods to read and write the most common file formats used for molecular structures (e.g. PDB, MOL2, HyperChem, etc.). The solvation component provides methods for calculating solvation effects. The first method that accounts for solvation is the atomic contact energy by Zhang et al. (1997). The second approach that we implemented is a numerical solver for the Poisson–Boltzmann equation. The structure component contains algorithms to search for common structural motifs in proteins, to map molecules onto each other, and to calculate solvent accessible and solvent excluded surfaces of molecules. For the visualization of the results, we designed BALLVIEW, a class hierarchy that visualizes BALL kernel objects with standard representations (ball and stick, van der Waals, ribbons, surfaces, etc.). The visualization was implemented using OpenGL and QT (Troll Tech AS, 1999). Hence, it is highly portable and enables the user to produce state-of-the-art graphical user interfaces with a minimum effort.

BALL has already proven its usefulness in the implementation of a quite complex algorithm for structure mapping (Becker, 1995). A first implementation of this algorithm in the course of Master’s thesis took about five months. Using BALL, we could implement the identical algorithm within a day (Boghossian et al., 1999). The implementation in BALL even outperformed the hand-coded algorithm significantly. This is mainly due to the carefully tuned data structures, the use of function inlining, and templates techniques.

BALL may be used free of charge for research and teaching. The source code is available from the BALL homepage (http://www.mpi-sb.mpg.de/BALL/). Commercial licenses can be obtained from algorithmic solutions software GmbH (http://www.algorithmic-solutions.com).

In the following section, we will describe the design and the implementation of BALL. In the following section, we will illustrate its potential for rapid software prototyping in a case study (the implementation of a complex energy function). In the final section, we summarize the state of BALL and point out some plans for future developments.

Design and structure of BALL

BALL is written in ANSI C++ (American National Standards Institute, 1998) and makes use of the Standard Template Library (STL, Musser and Saini, 1996). An overview of the structure of BALL is given in Figure 1. The foundation classes implement some basic data structures (e.g. strings, hash associative containers, and trees). The kernel which provides the molecular data structures (e.g. atoms, molecules), is based on these classes. Kernel and foundation classes are used to implement several independent
components, each covering a certain area of functionality. BALL has been tested with several compilers under Solaris, Linux, Tru64 Unix, and IRIX. Porting to other platforms should pose no special problems. It requires a C++ compiler that is sufficiently close to the ANSI standard.

The BALL kernel models biochemical real-world objects (e.g. atoms, molecules, proteins, etc.) with a class hierarchy. The fundamental concept used here is the composite design pattern (Gamma et al., 1995). Composite is the base class for all kernel objects. Each composite may contain an arbitrary number of other composites and implements basic methods to manipulate them. Thus, a tree of composites is formed. This data structure is extremely flexible and well suited to model the hierarchical relationships in biochemistry. For example a protein contains chains, chains contain residues, and residues contain atoms. Each of these biochemical entities has a corresponding kernel class (resp. Protein, Chain, Residue, and Atom) that are derived from Composite. This isomorphy between biochemical entities and the kernel classes result in an intuitive usability of the class hierarchy. The tree structure and the common base class lead to an open design that makes extensions very easy.

The foundation classes provide fundamental data structures and support classes used by the kernel and the basic components, but also by the applications. They can be grouped into data structures (e.g. strings, different types of trees, hash maps, grids, etc.), often used concepts (e.g. design patterns, object persistence, processors), system specific classes (e.g. filesystem and networking support), and mathematical objects (e.g. vectors, matrices, geometric objects). The total number of foundation classes is above one hundred, so we cannot discuss them in detail. However, we will present three important concepts used in the implementation of BALL.

The first concept is the processor concept. The processor concept is closely related to the visitor design pattern (Gamma et al., 1995). We will illustrate it in the following example. In some applications, it is necessary to calculate the center of mass of an molecular object. The obvious solution to this is the implementation of a method like getCenterOfMass() in the base class of all kernel objects. However, this method has two major drawbacks. First, it leads to a pollution of the base class interface. getCenterOfMass is certainly not the only interesting method, there are dozens more (the price of functionality). Second, adding new functionality changes the interface of all kernel classes and also requires a complete rebuild of the kernel classes. This is certainly not desirable. A processor avoids these disadvantages, because it allows the separation of the kernel data structures and the algorithms applicable thereon. All BALL kernel classes define a method apply which accepts processor objects as an argument. Each processor class defines three methods: start, finish, and operator(). First, the apply method calls start, a method which implements potential initializations of the processor. The kernel object is the root of a (sub)tree of kernel objects (composites). The apply method iterates over all nodes of this tree and calls the processor's operator() for each of these nodes. Finally, it calls finish to perform some final actions (like freeing data structures or calculating results). For our example, we define a class CenterOfMassProcessor.

The implementation of this class is trivial:

```
1  bool CenterOfMassProcessor::start()
2  {
3      mass_ = 0.0;
4      center_ = Vector3(0, 0, 0);
5      return true;
6  }
7
8  Processor::Result CenterOfMassProcessor::operator() (Atom& atom)
9  {
10     float atom_mass
11     = atom.getElement().getAtomicWeight();
12     center_ += atom.getPosition() * atom_mass;
13     mass_ += atom_mass;
14     return Processor::CONTINUE;
15  }
16
17  bool CenterOfMassProcessor::finish()
18  {
19    center_ /= mass_;
20    return true;
21  }
22
23  Vector3 CenterOfMassProcessor::getCenter()
24  {
25    return center_;  
26  }
```

Fig. 1. Overview of the structure of BALL.

![Overview of the structure of BALL.](image-url)
We can now apply this processor to BALL kernel objects (e.g. to a protein) by calling apply. The center of mass is returned by the member function getCenter:

```cpp
CenterOfMassProcessor proc;
protein.apply(proc);
Vector3 center_of_mass = proc.getCenter();
```

Another important concept is the Options class. Algorithms in computational molecular biology usually require a large number of arguments or parameters. The extension and the improvement of these algorithms therefore typically implies modifications in the interface. In order to prevent this, the options object may be used. It holds an arbitrary number of key-value pairs. Instead of changing the interface, additional parameters required in a modified algorithm are simply stored in this options object (an example for the usage of the options class is given in the following section).

The most important objects of BALL are persistent, that means, it is possible to store and retrieve them in arbitrary stream objects. This also includes SocketStream, a class that provides a C++ stream interface to TCP sockets. This feature is used, for example, to simplify the visualization: a stand-alone viewer application can visualize the results of any BALL application running on another machine in the internet. Hence, it is possible to run the visualization on a graphics workstation and perform the calculation on a cluster of machines or on a powerful compute server without graphics support. The results of the calculation can be sent to the visualizer with just a few lines of code. Persistence might also be used to create restart files for long-running applications.

The basic components of BALL cover a wide range of functionality. The file import/export component handles a number of standard formats used to store molecular structures. Since many different formats are used and a number of tools (e.g. BABEL, Walters and Stahl, 1997) exists to convert these formats, BALL offers only the most important ones, among others the PDB format and Tripos’ MOL2 format. However, the foundation classes provide basic support for the parsing of field-based and column-based formats to simplify the implementation of further formats. The Molecular Mechanics component provides generic force field classes that handle parameter files and parameter assignment. The generic force field contains a list of force field components. Each of these components represents a single contribution of the force field (e.g. bond stretching). For a number of commonly used energy terms, implementations are available. Thus, the realization of a new force field is reduced to the implementation of the missing interactions. A simple force field like CHARMM is implemented in a few days. The force field objects establish a basis for minimizer objects (steepest descent and conjugate gradient methods are already available) and molecular dynamics simulations. Using this generic force field, we implemented two standard force fields (AMBER and CHARMM).

Most force fields are well suited for the description of molecules in the vacuum, but do not account for solvation effects. Hence, BALL contains a solver for the PBE, that uses finite difference techniques to solve the PBE on a three-dimensional grid (for details, see the following section). The structure component provides methods to find common structural motifs in proteins, to calculate the solvent excluded/accessible surface of molecules, and to investigate structural similarities of proteins. Finally, the visualization component BALLVIEW can be used to visualize the results of the calculations. The usual representations for molecules such as ball-and-stick model, stick model, surfaces, and backbone representations are available. BALLVIEW is based upon OpenGL (or Mesa for non-OpenGL platforms) and QT, making it highly...
portable. There are two different ways to use BALLVIEW. First, it can be integrated into an application to provide a user interface and molecular visualization. However, constructing GUI-based applications is quite time-consuming, so BALLVIEW also provides a stand-alone viewer that can be integrated into BALL applications with just a few lines of code. The application then opens a TCP socket, connects to the viewer (that can run on a local or a remote machine) and transmits the data to the viewer. The appearance of the visualized models can be changed from the viewer as well as from the BALL application.

**Protein docking: a case study**

The Protein–Protein docking problem can be formulated as follows: given the three-dimensional structures of two proteins $A$ and $B$ that are known to form a complex $AB$, determine the structure of this complex.

Numerous approaches to this problem have been published in the last 10 years. Most of these algorithms work as follows: first, a structure generator creates a large number of potential complex structures. Then, the potential complex structures are evaluated. The evaluation process may consist of several filter steps that are applied successively (some docking algorithms use only a single filter). In each step, a fitness function or an energy function ranks all structures and removes the least promising candidates. In this way the number of potential structures is reduced step by step, while more and more sophisticated filters are applied. The first filter is usually a simple stability measure like the geometric complementarity. More sophisticated filters are generally more expensive (in terms of CPU time per structure evaluation) and therefore used in the final stages only, when the original number of structures has been significantly reduced. The docking algorithms usually generate a short list of the best candidates ranked with respect to the last energy function. If an approximation of the naturally occurring complex structure is on top of this list then the algorithm has successfully predicted the complex structure.

In the course of our protein docking project, we develop, implement, and test new functions for energy evaluation. We also compare these new functions with other well known methods. One of the methods that we implemented and tested, was proposed by Jackson and Sternberg (1995). It is a promising but complex model to predict protein–protein interaction energies. In this section we will first describe this model. Then, we will present a brief implementation in BALL that shows the rapid prototyping capabilities of BALL. Along with the description of the code, we will point out some features that simplify the application development.

Jackson and Sternberg provide a model for the estimation of protein binding energies that also takes into consideration the hydrophobic effect and the influence of the solvent on the electrostatic interaction. In earlier models, these two effects have often been neglected. The proposed model calculates the total free energy on binding $\Delta G_{\text{bind}}$ as a sum of different independent contributions:

$$\Delta G_{\text{bind}} = \Delta G_{\text{ele}} + \Delta G_{\text{cav}} + \Delta G_{\text{conf}} + \Delta G_{\text{vdW}}. \quad (1)$$

These contributions are the electrostatic free energy $\Delta G_{\text{ele}}$, the cavitation free energy $\Delta G_{\text{cav}}$, the change in the solutes’ conformational entropy $\Delta G_{\text{conf}}$, and the van der Waals free energy $\Delta G_{\text{vdW}}$. According to a previous study by Jackson and Sternberg (Jackson and Sternberg, 1994), $\Delta G_{\text{vdW}}$ can be neglected, because the new van der Waals contacts formed in the binding site are cancelled out by the contacts to the solvent lost on binding. Similarly, the change in side chain entropy may be neglected since it was found to be only a secondary effect.

The electrostatic part of the free energy on binding consists of three contributions:

$$\Delta G_{\text{ele}} = \Delta G_{\text{ele}}^A + \Delta G_{\text{ele}}^B + \Delta G_{\text{ele}}^{\text{int}}. \quad (2)$$

$\Delta \Delta G_{\text{ele}}^A$ and $\Delta \Delta G_{\text{ele}}^B$ describe the change in the solvation free energy associated with the binding process (the association of $A$ and $B$ leads to a change in the solvation free energy since the atoms of the binding site are no longer exposed to the solvent). $\Delta \Delta G_{\text{ele}}^{\text{int}}$ describes the electrostatic interaction between the proteins $A$ and $B$. The details of the calculation are explained further below.

The remaining contribution in equation (1), $\Delta G_{\text{cav}}$, describes the energy required to form a complex-shaped cavity in the solvent out of two cavities shaped like the proteins $A$ and $B$. The cavitation free energy is a linear function of the molecular surface area $A_{\text{MS}}$ of the cavity-forming molecule. Thus, we can express the change in the cavity free energy via the change of the molecular surface $\Delta A_{\text{MS}}$ during the binding:

$$\Delta A_{\text{MS}} = A_{\text{MS}}^{AB} - A_{\text{MS}}^A - A_{\text{MS}}^B \quad (3)$$

$$\Delta G_{\text{cav}} = \gamma_{\text{MS}} \Delta A_{\text{MS}} \quad (4)$$

$\gamma_{\text{MS}}$ is an empirical constant. Hence, the total free energy on binding can be calculated as follows

$$\Delta G_{\text{bind}} = \Delta \Delta G_{\text{ele}}^A + \Delta \Delta G_{\text{ele}}^B + \Delta \Delta G_{\text{ele}}^{\text{int}} + \gamma_{\text{MS}} \Delta A_{\text{MS}}. \quad (5)$$

The electrostatic component of the free energy is calculated using a so-called *continuum model*. This kind of model considers solvent molecules not explicitly but implicitly as a change in the dielectric constant: it assumes a solvent continuum. The solute is represented by a cavity of low dielectric constant in a high dielectric constant solvent (water). The boundary of the solute region is usually the molecular surface of the solute (solvent excluded surface).
Within this continuum model, the 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 \(\varepsilon(\vec{r})\):

\[
\nabla(\varepsilon(\vec{r})\nabla \phi(\vec{r})) - \kappa^2(\vec{r}) \sinh \left( \frac{\varepsilon(\vec{r})}{kT} \right) = -\frac{\rho(\vec{r})}{\varepsilon_0}. \tag{6}
\]

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

\[
\nabla(\varepsilon(\vec{r})\nabla \phi(\vec{r})) = -\frac{\rho(\vec{r})}{\varepsilon_0}. \tag{7}
\]

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 (Nicholls and Honig, 1991). The solution of the PBE 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:

\[
\Delta G_{\text{ele}}^{\text{total}} = \frac{1}{2} \sum_i q_i \phi(\vec{r}_i). \tag{8}
\]

The determination of the changes in electrostatic solvation free energy and the electrostatic interaction energy requires four Poisson–Boltzmann calculations (see Figure 3). First, the total electrostatic energy is calculated for \(A\) alone. On binding, protein \(B\) approaches \(A\). In the continuum model, \(B\) is represented as a region of low dielectric constant. Thus, we recalculate the total electrostatic energy of \(A\) and an uncharged \(B\)-shaped cavity of low dielectric constant (Figure 3a,b). The difference between the two values yields the change \(\Delta \Delta G_{\text{solv}}^A\) in electrostatic solvation free energy on binding for \(A\). The same calculation is then repeated for protein \(B\) (Figure 3c,d).

\[
\begin{align*}
\Delta \Delta G_{\text{solv}}^A &= \Delta G_{\text{ele}}^A, \text{cav}(B) - \Delta G_{\text{ele}}^A \\
\Delta \Delta G_{\text{solv}}^B &= \Delta G_{\text{ele}}^B, \text{cav}(A) - \Delta G_{\text{ele}}^B.
\end{align*} \tag{9,10}
\]

The interaction energy between \(A\) and \(B\) (or \(B\) and \(A\), which should yield the same value) is calculated as the energy of the charges of \(A\) in the field caused by \(B\) (or vice versa):

\[
\Delta G_{\text{int}}^{A-B} = \sum_{i \in A} q_i^A \phi_B(\vec{r}_i) = \Delta G_{\text{int}}^{B-A} = \sum_{i \in B} q_i^B \phi_A(\vec{r}_i) \tag{11}
\]

\(q_i^A\) is the charge of the \(i\)th atom of \(A\) and \(\phi_B(\vec{r}_i)\) is the potential at the coordinates of the \(i\)th atom of \(A\) caused by the charges of \(B\).

The Jackson–Sternberg model has been implemented in BALL and we will now explain the different parts of the BALL program. The code has been reduced to the minimum for the sake of brevity and comprehensibility. Thus, comments and include directives have been omitted.

First, the two proteins are read from two PDB files. Then, hydrogens are added, bonds are created, and the hydrogen positions are optimized using the AMBER force field. In the third step, radii and charges are assigned and the four Poisson–Boltzmann calculations are performed. Finally, the change in the solvent excluded surface area is calculated. It yields the cavitation free energy according to equation (4).

Reading PDB files is simple with BALL: create a PDBFile object, create a System that shall contain the molecules read from the file, and use the operator<< to read the file. This is performed twice to read the two proteins given as command line arguments. Tests for the validity of the arguments and a check whether the files exist have been omitted for brevity.

```c
int main (int argc, char** argv)
{
  system A, B;
  PDBFile f;
  f.open(argv[0]);
  f >> A;
  f.open(argv[1]);
  f >> B;
```

Adding hydrogens, optimizing them, assigning charges and radii is often a cumbersome process. In BALL this is performed by just a few lines of code. The most important object here is the fragment database (FragmentDb). This object contains templates for commonly used residues (i.e. all amino acids, nucleotides, and a number of frequently occurring hetero groups). This database also possesses a processor which examines each residue and adds missing hydrogens at positions derived from the template database (lines 14–15). Using another processor of the fragment database, the bonds that are required for a force field calculation are created (they are usually not contained in a PDB file). The hydrogen positions are quite reliable guesses, but for this application we need optimal positions, so we create an AMBER force field (Cornell et al., 1995) to optimize them (line 20). In line 23 we create an instance of Selector—a processor that selects Composites according to their properties. In this case, it selects all hydrogen atoms (\texttt{element(H)}). Then, we create a minimizer object (line 25) that uses a conjugate gradient method to optimize the positions of the selected atoms (options set in line 26). All other atoms will remain fixed.
With this minimizer the hydrogen atoms of A and B are optimized in lines 25 and 29.

13 FragmentDB fragment_db;
14 A.apply(fragment_db.add_hydrogens);
15 B.apply(fragment_db.add_hydrogens);
16 A.apply(fragment_db.build_bonds);
17 B.apply(fragment_db.build_bonds);
18
19 AmberFF amber;
20 amber.setup (A);
21 ConjugateGradientMinimizer minimizer(amber);
22 A.apply(H_selector);  // Select H atoms
23 minimizer.minimize();
24
25 amber.setup(B);
26 B.apply(H_selector);
27 minimizer.minimize();

The Poisson–Boltzmann calculation requires a bit of preparation. First, we have to assign the correct radii and charges (we use the PARSE set of charges and radii set derived by Sitkoff et al. (1994)). This data is stored in files containing an entry for each atom in each residue:

```
... 
ALA 2HB 1.0
ALA 3HB 1.0
ARG CA 1.7
ARG CB 1.7
...
```

This format is parsed by objects of the type AssignChargeProcessor and AssignRadiusProcessor. In lines 41–44, the correct charges and radii are assigned to A and B. The actual Poisson–Boltzmann calculation is performed by the FDPB object (Finite Difference Poisson Boltzmann). In order to get reasonable results, some parameters of the calculation are adjusted in lines 47–51. Each FDPB object contains options that describe the parameters of the calculation. One of these parameters is the grid spacing (set in line 49), another one is the dielectric constant of the solvent (which is set in line 47).

Lines 53–56 create a copy of B (named A_cavB) and clear all atom charges in this copy. This is done by the application of an instance of a ClearChargeProcessor in line 54. Then, a copy of A (tmp, created in line 55) is merged into A_cavB (using the splice method). This system now contains an uncharged copy of B and a charged copy of A, i.e. it contains A and an B-shaped cavity. The same procedure is performed for B in the lines thereafter.

```
32 AssignRadiusProcessor
33 radius_proc("PARSE.siz");
34
35 AssignChargeProcessor
36 charge_proc("PARSE.crg");
37
38 ClearChargeProcessor
39 clear_charge_proc;
40
41 A.apply(radius_proc);
42 B.apply(radius_proc);
43 A.apply(charge_proc);
44 B.apply(charge_proc);
45
46 FDPB fdpb;
47 fdpb.options.set
48 (FDPB::Option::SOLVENT_DC, "78.0");
49
50 fdpb.options.set
51 (FDPB::Option::SPACING, "0.6");
52
53 System A_cavB(B);
54 A_cavB.apply (clear_charge_proc);
55 System tmp(A);
56 A_cavB.splice (tmp);
57
58 System B_cavA(A);
59 B_cavA.apply(clear_charge_proc);
60 tmp = B;
61 B_cavA.splice(B);
```
In lines 63–65 the actual calculation is performed. A call to the setup method of FDPB creates the necessary data structures and fills them. The method solve then iteratively finds a solution of the PBE. The total energy is returned in line 65 and stored in the variable dG_sol_AcavB. Lines 67–69 repeat the same calculation for B and a A-shaped cavity. Finally, the total electrostatic energy of A and B are calculated in lines 71–79.

```cpp
63  fdpb.setup(A_cavB);
64  fdpb.solve();
65  float dG_sol_AcavB = fdpb.getEnergy();
66
67  fdpb.setup(B_cavA);
68  fdpb.solve();
69  float dG_sol_BcavA = fdpb.getEnergy();
70
71  float dG_int = 0.0;
72  fdpb.solve();
73  float ddG_solA = dG_sol_BcavA - fdpb.getEnergy();
74  float ddG_solB = dG_sol_AcavB - fdpb.getEnergy();
75
76  fdpb.setup(B);
77  fdpb.solve();
78  float ddG_solB = dG_sol_BcavA - fdpb.getEnergy();
79  float ddG_solA = dG_sol_AcavB - fdpb.getEnergy();
```

The calculation of the interaction energy concludes the calculation of the electrostatic component. The macro BALL_FOREACH_ATOM is used to iterate over all atoms of B and to calculate the sum in equation (11). The electrostatic potential \( \phi(\vec{r}) \) is contained in the member variable phi_grid of fdpb. The potential \( \phi(\vec{r}) \) at the atom center \( \vec{r} \) is obtained as the linear interpolation of the eight surrounding grid points. This interpolation is performed by the method getInterpolatedValue. The resulting energy is then scaled in line 77 to units of kJ/mol (BALL internally uses kJ/mol as energy unit).

```cpp
81  float dG_int = 0.0;
82  AtomIterator atom_it;
83  BALL_FOREACH_ATOM(A, atom_it)
84  {
85      dG_int += atom_it->getCharge()
86          * fdpb.phi_grid->getInterpolatedValue
87          (atom_it->getPosition());
88  }
89  dG_int *= Constants::AVOGADRO;
```

The solvent excluded surface (SES) is calculated using the algorithm of Connolly (1983). In lines 96–98 the molecular surface areas of \( A \), \( B \), and the complex \( AB \) are calculated, then in line 99 the change in cavitation free energy is calculated according to equation (4). Finally, in lines 101–103 the resulting energy is calculated and printed.

```cpp
93  const float gamma
94      = 0.069 * Constants::JOULE_PER_CAL;
95
96  float A_A = calculateSESArea(A);
97  float A_B = calculateSESArea(B);
98  float A_AB = calculateSESArea(A_cavB);
99  float dG_cav = (A_AB - A_A - A_B) * gamma;
100
101  cout << "Total_free_energy_on_binding: ";
102  cout << ddG_solA + ddG_solB  
103      + dG_int + dG_cav << "kJ/mol" << endl;
104 }
```

**Performance**

To give an impression of the performance of BALL, we give benchmarks for the two main steps in the Jackson–Sternberg model. The first benchmark is the FDPB calculation. We used a grid of 140×140×140 points and calculated the electrostatic energy for a medium sized protein (1TPO from the PDB (Bernstein et al., 1977)). We compared the running time (in CPU seconds, including the setup of the grid) of BALL with the running time of Delphi ((Nicholls et al., 1990), the software used in the original paper by Jackson and Sternberg) on the same machine and with the same settings. In both cases, 220 iterations of the linear PBE were required to achieve convergence.

Our second benchmark is the optimization of all hydrogen atoms (1594 atoms) in the same protein using the AMBER force field and a conjugate gradient minimizer. The optimization was stopped when the RMS gradient fell below 1 kJ/(mol Å). In this benchmark, we compared our code with the AMBER implementation of HyperChem 4.5/SGI (Hypercube Inc., 1995).

Both benchmarks were executed on an SGI Octane (768 MB RAM, 195 MHz R10 000 processor). The results are given in the following table:

<table>
<thead>
<tr>
<th>Method</th>
<th>CPU time (s) BALL</th>
<th>CPU time (s) Delphi/HC</th>
<th>Relative performance of BALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDPB</td>
<td>191</td>
<td>216</td>
<td>113%</td>
</tr>
<tr>
<td>AMBER</td>
<td>501</td>
<td>311</td>
<td>62%</td>
</tr>
</tbody>
</table>

Since the FDPB code of BALL was carefully hand-tuned, we are able to outperform the Fortran implementation of Delphi. In the molecular mechanics code, our implementation loses performance against the HyperChem implementation since we have a reduced locality in our data and the AMBER code was not yet tuned to yield optimal performance. Nevertheless, this loss in performance seems acceptable when compared to the reduced development times.
Conclusion

Using modern software engineering techniques the development times for software in computational molecular biology can be drastically reduced and the software quality can be increased. We have shown that BALL provides a useful tool for this purpose. We intend to develop BALL towards a public repository for data structures and algorithms in the field of computational molecular biology. BALL helps the developer to concentrate on the actual problem instead of reimplementing code again and again. Since the resulting applications can be freely distributed, the propagation of new methods will be accelerated and the software quality and their impact will be increased. Algorithms for sequence alignment, threading, and the prediction of nuclear magnetic resonance spectra will be implemented in the near future. Besides adding new functionality, we also intend to port BALL to new platforms.

Acknowledgements

This work was supported by the DFG research cluster ‘Informatikmethoden zur Analyse und Interpretation großer genomischer Datenmengen’, grant LE952/2-1.

References


