Structure-based bioinformatics with BiochemicalAlgorithms.jl

Jennifer Leclaire[1], Thomas Kemmer[1], and Andreas Hildebrandt[1]

[1]Scientific Computing and Bioinformatics, Institute of Computer Science, Johannes Gutenberg-University Mainz

ABSTRACT

Keywords

Julia, Structure-based bioinformatics, C++, BALL

1. Introduction

The aim of structural bioinformatics is the analysis and targeted manipulation of three dimensional structures of biological macromolecules such as proteins and nucleic acids (RNA/DNA). This research area combines and applies knowledge from diverse disciplines ranging from fundamental physical laws such as Newton's equation of motion to complex situations requiring in-depth knowledge of biochemistry and advanced numerical computing methodologies.
More precisely, molecular modeling techniques typically rely on molecular mechanics in which a molecular force fields is being used to compute the energy of the examining structure. Resulting applications include structure minimization, molecular dynamics (MD) simulations and molecular docking scenarios. The latter is inevitable for drug design because it computes the best configuration for a given protein and a ligand (the drug) and compute the binding affinity. The importance of molecular applications has been demonstrated during the Covid-19 pandemic. Tools for molecular modeling and in particular docking suites were attracting widespread interest again in order to model SARS-COV-2 proteins and running molecular docking simulations.

Write about: Rational Drug Design economically important

Why do we have to run simulations on the computer? Why is it necessary to perform it on a computer? Example Protein Ligand Docking... As an example.
For more than 50 years researcher have been studying molecular functions of proteins. The first attempts date back to 1959, when Max Perutz and John Kendrew used X-ray crystallography to to solve the structure of hemoglobin.
A vital aspect of this research field is the availability of molecular structure data, which was solved by the initialization of the protein data bank in 1972. Although it only contained 7 structures, it now provides 227.561 experimentally solved structures (dated on:). With the rapid rise of computed structures

check number of structures in pdb

software development in interdisciplinary research areas such as structural bioinformatics was, and still is, typically challenging. Software packages for handling molecular structures and molecular applications are available for many years and can be rougly divided into open-source and closed-sourced tools. Most software packages were created from 1995 to 2010 and written in C++. Although this choice of programming language enables the required efficiency, it does not allow for rapid prototyping, which is why some software packages provide an additional interface in a scripting language like Python.

Cite

The tools were often only designed for one specific task e.g., implementation of a structure minimization algorithm, docking algorithm... One example for the latter is Schroedinger [5]

cite single approache

cite schroed

—docking algorithmus
—kraftfelder..

The development of software packages for structural bioinformatics remains a challenging task.
Several packages already exist in Julia related to structural bioinformatics. Most prominently, the packages under the two Github communities *Molecular Simulation in Julia* and *BioJulia*, which puts an emphasis on sequential bioinformatics.
Molly.jl is an excellent package for molecular simulations written in Julia and is part of the *Molecular Simulation in Julia* Github community [3]. Additionally, *ProtoSyn.jl* is an interesting approach to handle and manipulate oligopeptides but does not seem to be actively maintained any more (the last push is 10 months ago).
A platform from which molecular file formats can be read and write, the entire preprocessing pipeline can be integrated and the infrastructure for molecular mechanics are provided is still lacking. There remains a need for a basis from which software packages for handling molecular file formats and the proper preprocessing
To the best of our knowledge a comparable package for molecular analysis does not exist in Julia. Furthermore, the ongoing developments around the BALL project including the molecular viewer indicate a strong need for such a framework.
Here, we present BiochemicalAlgorithms.jl . We provide the basis for interesting analysis encompassing the entire molecular modeling pipeline:

—Reading common data formats such as PDB, hin, mol and JSON
—Preprocessing the input by preparing the entire system ready to simulate.
—Molecular Mechanics such as AMBER ForceField
—(Output writing) such as JSON

BiochemicalAlgorithms.jl is designed to be a platform from which other packages can be included.

2. BALL - Biochemical Algorithms Library

As already mentioned, the main intention for the development of BALL as well as for BiochemicalAlgorithms.jl is to generate a

Table 1. : Desing goals of BALL

Ease of use	Robustness
Openness	Functionality

framework for rapid prototyping of molecular applications. This section summarizes the key concepts of BALL that motivated the design of our project [1]

The initial work on the BALL project started in 1996, resulting in the C++-written library BALL and its accompanying molecular viewer, *BALLView*. One reason for the success of BALL is its sophisticated design; it is an object-oriented project with four design goals as shown in Table 1.

The object-oriented approach facilitates ease of use in combination with the well documented and intuitive interface. As can be seen in Figure 1 BALL is structured in several layers. On top of the standard template library (STL), resides the foundation classes, providing a set of general data structures such as hash sets and mathematical objects (e.g. matrices, vectors,...). The third layer is the actual core of the library: the `KERNEL` classes, which contain data structures for molecular entities. The basic components represent fundamental functionalities and are placed on top of the core, with the exception of the visualization module that is based on QT and Open GL. Finally, the application layers can be used to develop own applications or use the available tools.

Hildebrandt et al. published an updated version in 2010, featuring the CMake-Buildsystem and Python bindings. These additions improved usability and openness, allowing easier integration of external packages and increased portability to other compilers and operating systems.

BALL 's uniqueness stems from its rich functionality integrated in a single easily extensible open-source platform. Figure 2 shows the kernel and its classes are forming three different frameworks: the general molecular framework, the protein and the nucleic acid frameworks. All kernel classes are implemented through the realization of a composite pattern. More precisely, the composite class is the base class for all derived classes representing molecular entities such as `Atom`, `Protein`, etc. or container classes `System`, `AtomContainer`, and so on. Based on these three frameworks, BALL provides functionalities for various different steps in molecular analysis ranging from tools for preprocessing such as file import and export, addition of missing atoms, normalizing name schemes, to complex molecular structure analysis (energy minimization, mapping) and advanced solvation methods. These implemented applications are well-tested and validated, ensuring robustness.

BALL 's well-designed and structured nature has contributed to its long-lasting popularity, with one of the largest user communities for open-source software in its field.

3. A fresh approach for Structural Bioinformatics: BiochemicalAlgorithms.jl

In this work, we sought to redesign the popular BALL package for molecular analysis and simulation. This section initially examines reasons for a redesign in Julia, followed by a description of the core implementation of BiochemicalAlgorithms.jl closing with the benchmark results.

[1] An in-depth description of the entire BALL framwork is beyond the scope of this article. Confer the main publications [5, 4] for more details.

3.1 Reasons for a redesign or *why Julia?*

While the design goals and the need for an open-source framework with the rich functionality such as BALL 's is still present, the realization of the main design principles of BALL are highly dependent on the choice of the programming language. From today's perspective in particular with regard to its purpose as a platform for rapid application development (RAD), the usage of C++ may be considered suboptimal.

As for many scientific software packages, the development times for applications play a crucial role for the acceptance and usability of the underlying software. For example, a great deal of time may have to be spend by installing the library with its dependencies. Although the CMake build system has been integrated in version 1.3, setting up the library is a highly non-trivial task. Additionally, the development times are massively influenced by the knowledge of the used programming language. As a low-level language, C++ is known to require more time to be learned compared to scripting languages such as Python [6]. Even with the additional Python bindings, the integration of new functionality is still not straightforward. In contrast, the implementation of new features is typically associated with the addition of massive amounts of boilerplate code. This applies to an even greater extent, in cases where portability to different platforms and compiler settings have to be supported.

Consequently, BALL itself can be considered as a textbook example for the two language problem, which is very common for scientific computing project. In the latter, the core functionality is often implemented in a low-level programming language, ensuring the required performance, while higher-level programming languages are used for user-friendly interfaces to the core functionalities. Julia is exactly developed for these situations [2, 7].

Nevertheless, it is important to keep in mind, that back in 1996 and still in 2010, C++ was the best choice for the implementation of BALL .

The design goals as described in the previous section are achieved in a much easier way with the usage of Julia:

—ease of use: Julia is easier to learn and ideas can be implemented from scratch. The times to inter, installation and portability, documentation is integrated

—Openness: integration of external tools trivial,

—robustness: BiochemicalAlgorithms.jl has been developed with accompanying test case for the core structures as well as for the functionalities, benchmarks have been implemented to ensure performance, both things were integrated on the fly

—functionality, not yet feature complete but we can already do things as described in the applications section

3.2 The core

In addition to the four original design goals, we sought to make BiochemicalAlgorithms.jl comparable in performance to BALL . The core data structures play key roles in the efficiency of the entire framework. When working with structural data of molecular entities such as proteins, or nucleic acid molecules (DNA/RNA), the representation of three dimensional coordinates play an important role.

Just like in BALL we define a similar hierarchical structure...as can be seen in fig.. Optimized for random access to any atoms... same hand easy to use for Julia users or Python users familiar with Pandas DataFrames... So initially, representation was based on DataFrames. While theis is extremely intuitive, it has some

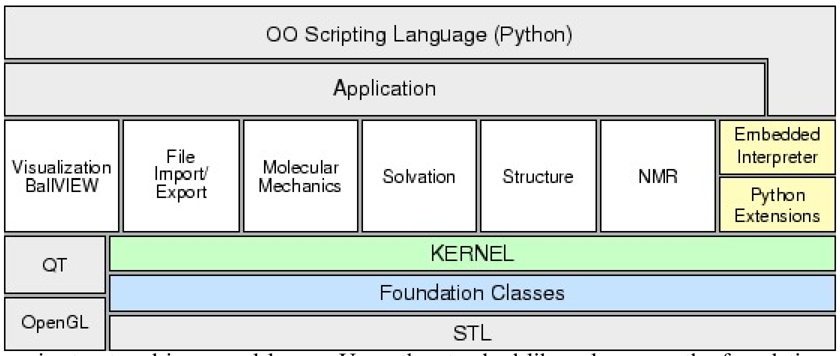

Fig. 1: BALL 's architecture is structured in several layers. Upon the standard libary layer are the foundation classes and ontop of them the kernel. Several module extend the interface for visualization, file import and expoert, molecular mechanics, solvation, structure and NMR. The C++ written frameworkis extended by Python interface for fast scripting purposes. The figure was taken from the official BALL documentation [1].

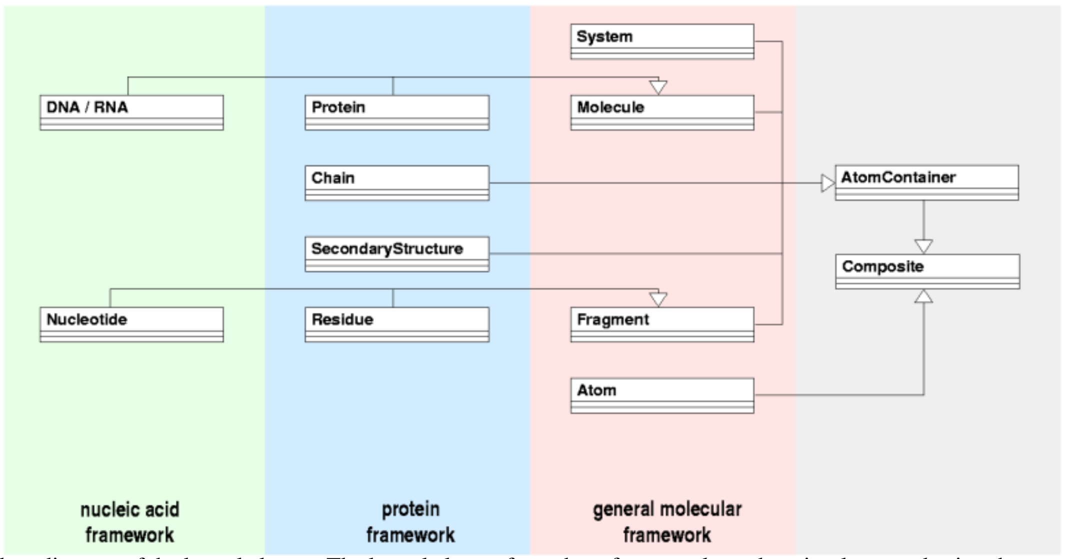

Fig. 2: UML class diagram of the kernel classes. The kernel classes form three frameworks and are implemented using the composite pattern. The figure was taken from the official BALL documentation [1].

Table 2. : Example tasks and the required time

Description	BALL	BiochemicalAlgorithms.jl
tba	tba	tba
hline tba	tba	tba
tba	tbal	tba

3.3 Benchmarks

On par with its C++ predecessor it is intuitiv to write code that is

4. Applications

The functionality and usability of BiochemicalAlgorithms.jl is best demonstrated with examples. We chose three small scenarios, we begin with a simple example to show how some of the core structures can be created and used. Then we had over to a more complex example where we want to compare two different configurations of the same molecule. Finally, we want to demonstrate the elegance of Julia written code in comparison to C++. In the last application, we briefly introduce the accompanying visualization tool BiochemicalVisualization.jl that has been developed alongside the BiochemicalAlgorithms.jl framework.

4.1 Generating a water molecule

The class diagramm in Figure 3 shows the core of BiochemicalAlgorithms.jl . The interface is intuitively designed and the interactions with the components are straightforward as can be seen in code listing 1. The center of an application is the `System`. If no such system is explicitely created a default system is generated. Atoms can be created and the corresponding bonds and will automatically be part of the defined system. The names of the classes for the molecular entities and related functionalities were carefully chosen to be as intuitive as possible. The resulting system with the contained water molecule can be visualized via the visualization tool BiochemicalVisualization.jl . See Figure 4 and the following sections for more details.

Code 1: Intuitive usage of BiochemicalAlgorithms.jl core components

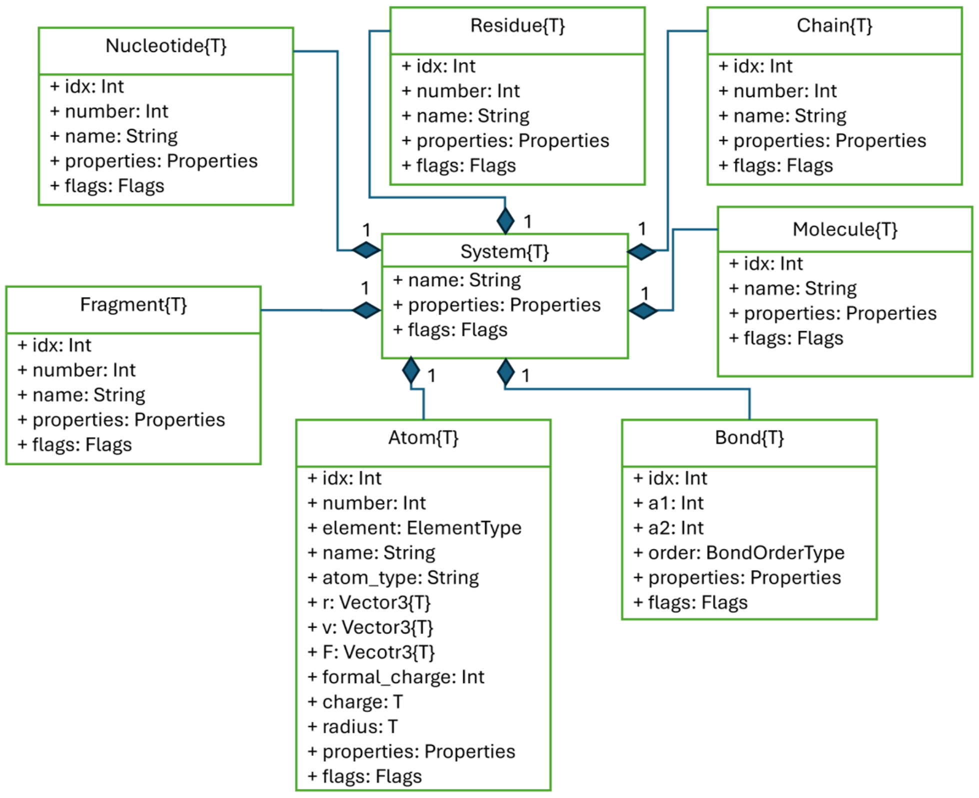

Fig. 3: UML-Diagramm of the core of BiochemicalAlgorithms.jl . In the center resides the `System` interface. All other functionalities are grouped around that core piece. Only the most important functionalities of each class are shown.

```
using BiochemicalAlgorithms
using BiochemicalVisualization

sys = System()
h2o = Molecule(sys)

o1 = Atom(h2o, 1, Elements.O)
h1 = Atom(h2o, 2, Elements.H)
h2 = Atom(h2o, 3, Elements.H)

h1.r = [1, 0, 0]
h2.r = [cos(deg2rad(105)), sin(deg2rad(105)), 0]

Bond(h2o, o1.idx, h1.idx, BondOrder.Single)
Bond(h2o, o1.idx, h2.idx, BondOrder.Single)

println("Number of atoms: ", natoms(h2o))
println("Number of bonds: ", nbonds(h2o))

ball_and_stick(sys)
stick(sys)
van_der_waals(sys)
```

4.2 RMSD computation and Application of Amber

A very common task in structural analysis is the comparison of two or more structures. The following example will demonstrate the entire molecular pipeline. First, the two pdb files are loaded into a system container. Compared to the previous example, the variable `sys` represent a `Vector` of `Systems` instead of a single system. The systems are preprocessed with the `FragmentDB`, a database containing known fragments of molecules. The preprocessing steps include the normalization of different naming standards, the reconstruction of missing parts of the molecules and the creation of bonds, since pdb format usually contain no or incomplete bond information. After the preprocessing, the molecular structures are each applied to a molecular force field and here the amber energy of the systems are computed. The structures studied in this example are different configuration of the same molecule and can be mapped onto each other. Before and after the mapping the RMSD is computed and displayed.

This example demonstrates the rich functionality of BiochemicalAlgorithms.jl by just a few lines of code. The steps that were carefully taken to prepare the systems are shown in Figure5.

Code 2: Comparison and mapping of two similar structures

```
sys = load_pdb.(["data/arnd1.pdb",
                 "data/arnd2.pdb"])

fdb = FragmentDB()
normalize_names!.(sys, Ref(fdb))
reconstruct_fragments!.(sys, Ref(fdb))
build_bonds!.(sys, Ref(fdb))

println.(sys)

compute_energy.(AmberFF.(sys), verbose=true)

println("RMSD before mapping: ",
        compute_rmsd(sys[1], sys[2]))

map_rigid!(sys[1], sys[2])

println("RMSD after mapping: ",
        compute_rmsd(sys[1], sys[2]))
```

output?

4.3 A trivial comparison between C++ and Julia

The verbose nature of C++ comes apparent in a simple task. A typical situation in molecular simulation is to find out, which atoms are in a certain proximity of each other. As these atoms can exert interactions, which are important for the stability of a structure configuration. We would solve this task with advanced data structures such as a hash grids. However, for simplicity, we consider the following task: We want to count the contacts between two separate molecules, that are in close proximity. We will define a contact, if the distance between two carbon atoms C_β atoms is smaller than 6Å.

The code listing below shows the solution for the task in C++. It is important to note here, that due to readability, the necessary includes for this even short code snippet are not shown. Using two nested for-loops, possible C_β atoms are searched, whose distance from each other is computed in the next step.
In contrast, with BiochemicalAlgorithms.jl this task is solved much

4.4 Visualization using BiochemicalVisualization.jl

A key feature of BiochemicalAlgorithms.jl is the visualization tool *BiochemicalVisualization.jl* that has been developed alongside the main framework. As shown in Figure 4 BiochemicalVisualization.jl currently supports three different representation of atomic structures, namely `ball-and-stick`, `van-der-Waals` and `stick` (cf. code listing 1).
When dealing with three dimensional structures of macromolecules, visualization plays an important role for supporting the development of insights into molecular functions. The possibility to visualize and interactively modify the representations provides great support during modelling scenarios. For instance, the tool has been used to visualize different steps from code listing 2. The first image left represent the raw input read from the underlying pdb file, the image in the middle shows the same input after preprocession it with the fragment data base (lines4-7). Finally, the mapping of both structures is shown in the image on the left. As can be seen, the structures do not match perfectly onto each other.
Even this rather simple example already demonstrates the advantage of a visual representation that can be modified and manipulated in context of modelling scenarios and how the visualization supports the development of knowledge of molecular functions.

5. Conclusion

6. References

[1]

[2] Julia Community. Two language problem. what is it? `https://discourse.julialang.org/t/two-language-problem-what-is-it/82925`, 2023. Accessed: December 13, 2024.

[3] Joe G Greener. Differentiable simulation to develop molecular dynamics force fields for disordered proteins. *Chemical Science*, 15:4897–4909, 2024.

[4] Andreas Hildebrandt, Anna Katharina Dehof, Alexander Rurainski, Andreas Bertsch, Marcel Schumann, Nora C. Toussaint, Andreas Moll, Daniel Stöckel, Stefan Nickels, Sabine C. Mueller, Hans-Peter Lenhof, and Oliver Kohlbacher. BALL - biochemical algorithms library 1.3. *BMC Bioinformatics*, 11(1):531, October 2010. doi:10.1186/1471-2105-11-531.

[5] Oliver Kohlbacher and Hans-Peter Lenhof. BALL—rapid software prototyping in computational molecular biology. *Bioinformatics*, 16(9):815–824, September 2000. doi:10.1093/bioinformatics/16.9.815.

[6] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[7] Julia Data Science. What julia aims to accomplish? - the two-language problem. `https://juliadatascience.io/julia_accomplish`, 2023.

Code 3: The resulting C++ code for the example task consist of a lot of boilerplate code.

```cpp
int count_contacts(const AtomContainer& ac1, const AtomContainer& ac2, double thres = 6.0) {
    auto contacts = 0;
    for(auto ait1 = ac1.beginAtom(); +ait1; ++ait1) {
        if(ait1->getName() != "CB")
        continue;

        for(auto ait2 = ac2.beginAtom(); +ait2; ++ait2) {
            if(ait2->getName() != "CB")
            continue;

            auto dist = ait1->getPosition().getDistance(ait2->getPosition());
            if(dist <= thres) {
                contacts++;
            }
        }
    }
    return contacts;
}
```

Code 4: The resulting Julia code for the example task is much more elegant.

```julia
using BiochemicalAlgorithms

function filter_cbeta(ac::AbstractAtomContainer{Float32})
    (atom for atom in atoms(ac) if atom.name == "CB")
end

function is_in_contact(r1::Vector3{Float32}, r2::Vector3{Float32})
    distance(r1, r2) <= 6
end

function count_contacts(ac1::AbstractAtomContainer{Float32}, ac2::AbstractAtomContainer{Float32})
    count( t -> is_in_contact(t...), ((a1.r, a2.r) for a1 in filter_cbeta(ac1), a2 in filter_cbeta(ac2)))
end
```

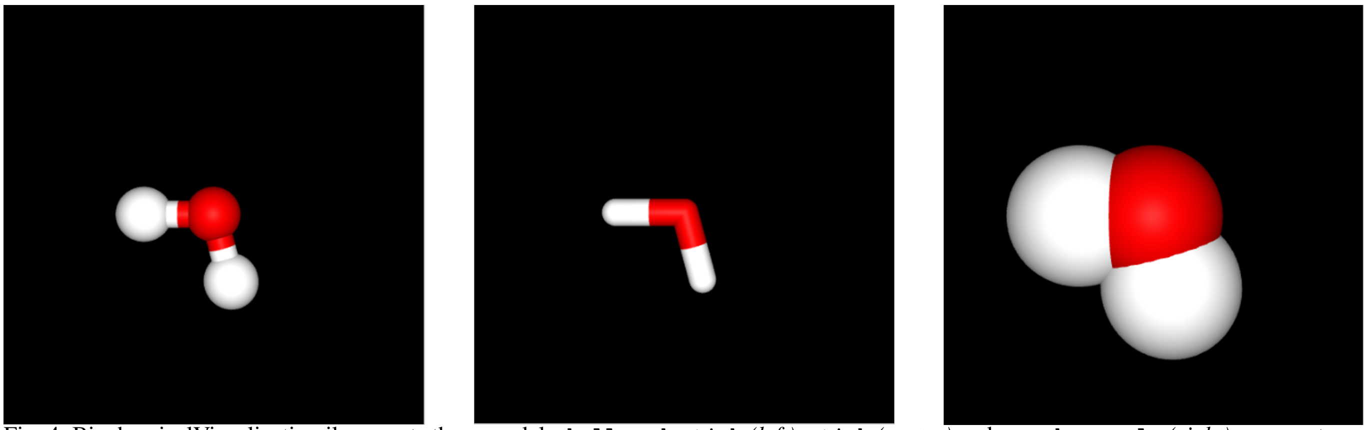
Fig. 4: BiochemicalVisualization.jl supports three models: `ball-and-stick` *(left)*, `stick` *(center)* and `van-der-waals` *(right)* representation of the water molecule as generated by the code listing 1.

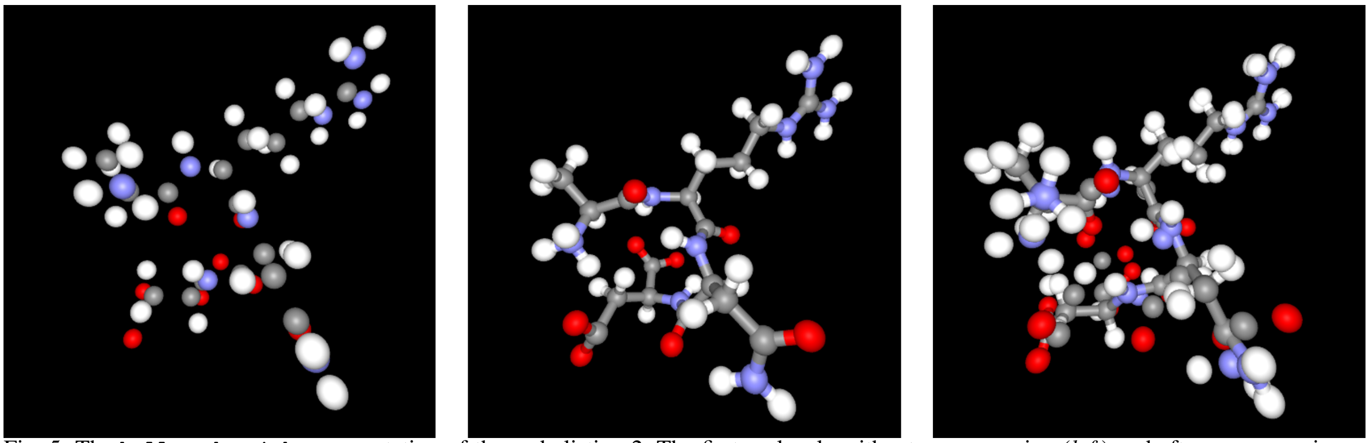

Fig. 5: The `ball-and-stick`-representation of the code listing 2. The first molecule without preprocessing (*left*) and after preprocessing (*center*). Finally, the two structures are superposed (*right*).