juliacon

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 importance of these molecular applications has been demonstrated during the Covid-19 pandemic. Tools for molecular modeling and in particular docking suites were attracting widespread interest: First, the structure of virus proteins were predicted and examined including analysis of the effects of mutations originating from various virus strains. The knowledge of these molecular functions was then used in the context of rational drug design for the purpose of finding a potential drug for the treatment of Covid-19.

Another vital aspect of this research field has been demonstrated through the Covid-19 pandemic, which is the availability of molecular structure data. The availability of experimentally resolved structures was limited for many years leading to a slowdown of the progress in this area. In recent years, the number of experimentally structures is increasing dramatically through advances in electron microscopy and with the rapid rise of computed structures, the availability no longer restrain the development of structural bioinformatics applications.
Software development in interdisciplinary research areas was, and still is, typically challenging. Software packages for handling molecular structures and molecular applications are available for many years and can be divided into open-source and closed-sourced tools. One example for the latter are the tools provided by schroedinger:`ProteinPreparationWizard` deals with the preprocessing of protein structural models including missing atoms, missing hydrogens, bond order assignment etc. or `LiveDesign` for the docking and design of ligands. Although Schroedinger provides entire platforms for several task, these have the indisputable disadvantage of being closed-source. Most open-source software packages were created roughly from 1995 to 2010 and were often only designed for one specific task e.g., implementation of a structure minimization algorithm, docking algorithm...
An exception to this was the introduction of Biochemical Algorithms Library (BALL) by Kohlbacher et al. in 1996. BALL is a well designed framework for molecular structure analysis providing a rich functionality namely file import and export of most common file formats, structure preprocessing, molecular mechanics, advanced solvation methods and visualization options. BALL used to have one of the biggest user communities in this field as can be assumed from the long lasting maintenance and ongoing development. In 2010 a new version was published featuring Python bindings for rapid application development (RAD).
Like BALL , more recent software packages were also written in C++ with additional Python bindings [**?**]. While C++ is necessary for the required efficiency of the programms, it effectively hinders the rapid prototyping of molecular algorithms.

cite single approache

While the development of software packages for structural bioinformatics remains a challenging task, the choice for the programming language is not. In contrast, Julia combines the efficiency and numerical stability required for molecular simulations with the possibility of rapid development.

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 [**?**]. 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 devel-

Table 1. : Desing goals of BALL

Ease of use	Robustness
Openness	Functionality

opments 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 analysis studies 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 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.

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

3. 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.

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 sub optimal.
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 [**?**]. 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 and guarantees the numerical stability and accuracy required for molecular mechanics applications [**?**, **?**].
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 .
Switching our development from C++ to Julia has greatly simplified conforming the design goals denoted in Table 1:

—Ease of use: BiochemicalAlgorithms.jl 's source code provides a better readability as the usage of Julia does not produce so much boilerplate code compared to BALL . The integration of documentation, basic tutorials and test cases facilitates the introduction to BiochemicalAlgorithms.jl , not to mention the trivial installation via Julia's Package manager.

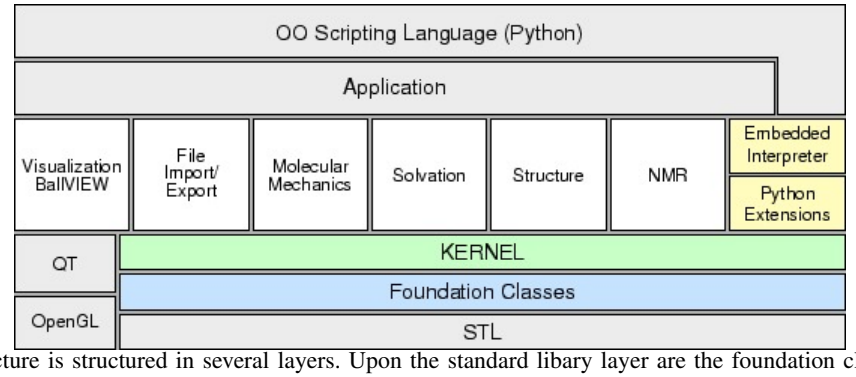

Fig. 1: BALL 's architecture is structured in several layers. Upon the standard libary layer are the foundation classes and ontop of them the kernel. Several module extend the interface for visualization, file import and expoert, molecular mechanics, solvation, structure and NMR. The C++ written frameworkis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

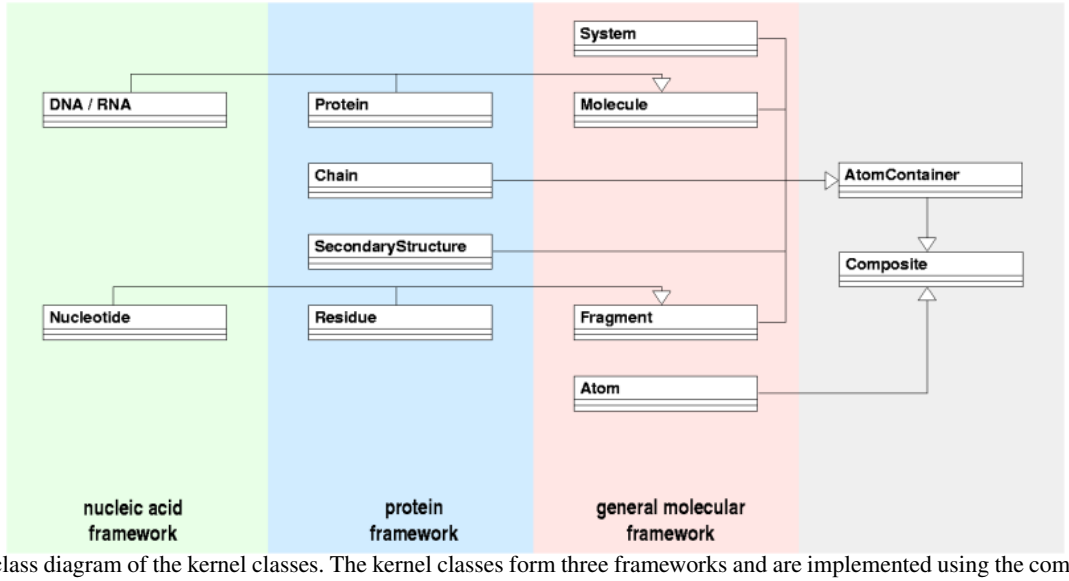

Fig. 2: UML class diagram of the kernel classes. The kernel classes form three frameworks and are implemented using the composite pattern. The figure was taken from the official BALL documentation [**?**].

—Openness: Just like the installation, the integration of external tools to BiochemicalAlgorithms.jl is straightforward. Our well documented framework allows the integration of own applications seamlessly.

—Robustness: One of the strength of Julia is the integrated unit testing functionality allowing to test implemented code on the fly. BiochemicalAlgorithms.jl has been carefully developed with accompanying test case for the core structures as well as for the functionalities ensuring non-faulty behavior using `TestItemRunner.jl`[**?**]. Benchmark test cases are implemented in order to assess performance of typical tasks with the help of `BenchmarkTools.jl` and `Pkgbenchmark.jl` [**?**, **?**]. The results of the benchmark study are described in section below.

—Functionality: BiochemicalAlgorithms.jl implements a set of standard data structures for molecular entities and already provides different functionalities based on these including import of structures stored in PDB, pubchem or hin files, molecular mechanics more precisely an interface for force fields and an Amber implementation, structure minimization algorithms. The interface are designed in a way that facilitates adoption e.g., implementation fo an own force field or changing an optimizer for the structure minimization.

3.2 The core representation

In addition to the four original design goals, we sought to make BiochemicalAlgorithms.jl comparable in performance to BALL . When working with structural data of molecular entities such as proteins, or nucleic acid molecules (DNA/RNA), the representation of atoms play an important role.
More precisely, we decided to put the `System` at the center of every application in BiochemicalAlgorithms.jl . If the system is not defined explicitly, it will be generated by default. As shown in Figure 3, the system contains data structures for the representation of atoms, bonds, molecules, chains, residues, nucleotides and fragments as soon as they are generated explicitly (see code listing 1) or populated by reading files (see code listing 2).

Table 2. : Example tasks and the required time

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

The representation of an atom with their position, velocity and force contributes substantially to the efficiency of the entire framework. Therefore, careful attention has been given to the implementation of the underlying data structure. Due to its popularity and intuitive usage, a preliminary attempt consisted of a representation using `DataFrames.jl` [**?**]. However, we decided to move to a costume implementation of the `Tables.jl` interface [**?**]. The costume implementation enabled more flexibility regarding the design of the interface and in addition, initial benchmarks indicated a better performance (data not shown).

Why is more flexibility needed?

The Table 2 lists the results of a benchmark study comparing the time needed for typical core functionalities in BALL and BiochemicalAlgorithms.jl . It is evident BiochemicalAlgorithms.jl is on par with its C++ predecessor.

Tables interface allows to be converted to BioStructures.jl stuff and so on

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 0**??** and the following sections for more details.

Code 1: Intuitive usage of BiochemicalAlgorithms.jl core components

```
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 Figure**??**.

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 RAD in BALL and BiochemicalAlgorithms.jl

Rapid application development is a key feature of the BiochemicalAlgorithms.jl package. In the following, we show a comparison between BALL and BiochemicalAlgorithms.jl for a simple task revealing the verbose nature of C++.
A typical situation in molecular simulation is to find out, if atoms are in a certain proximity of each other. This is of interest, because these atoms can exert interactions, which are important for the stability configuration. However, we consider a simplified definition of the problem: 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

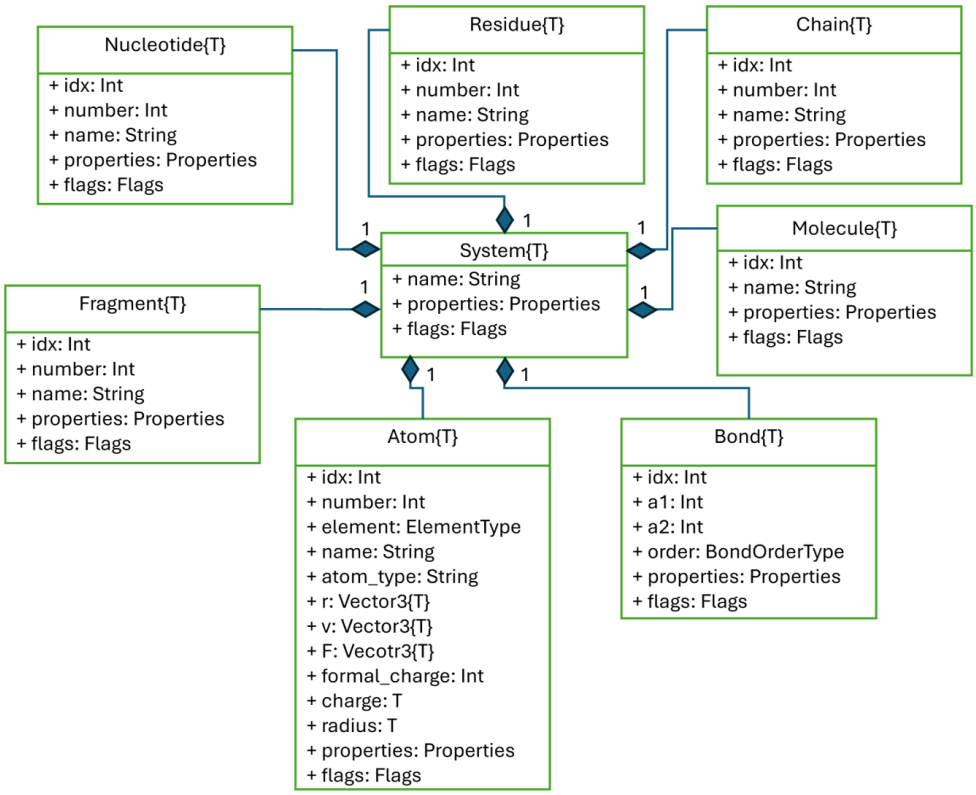

Fig. 3: UML-Diagramm of the core of BiochemicalAlgorithms.jl . In the center resides the `System` interface. All other functionalities are grouped around that core piece. Only the most important functionalities of each class are shown.

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.
Although the code is functional, it demonstrates the verbosity if C++ compared to the solution in Julia **??**. Here, two functions are created serving for the filtering of the molecules and for the computation of the distances of two atoms. With these two, the actual function for the counting consists only of a single line of code. This examples showcases the elegance of the BiochemicalAlgorithms.jl framework compared to BALL . More advanced data structures such as hash grids would be more efficient for larger systems, but this example serves to illustrate the basic approach.

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 **??** 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

We have presented a single and modern platform for the rapid application development of molecular structure analysis and simulations. While other Julia packages focused either on a single task like the manipulation of peptides or molecular dynamic simulations, BiochemicalAlgorithms.jl serves as framework for exploration studies of structure or the proper initialization of structures for molecular simulations through the usage of the fragmentDB. Like its C++ predecessor, BiochemicalAlgorithms.jl provides a robust but flexible core with additional functionalities. More precisely, the core has been carefully designed using a costume realization of the `Tables.jl`-interface. Thereby, our platform

—can be integrated to other tools using the tables interface
—fits into julia mol sim or BioJulia

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

filter_cbeta(ac) = (atom for atom in atoms(ac) if atom.name == "CB")
is_in_contact(r1,r2) = distance(r1,r2) <= 6

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
```

—Achievement: BiochemicalAlgorithms.jl allows the rapid prototyping of molecular analysis

—Meaning of achievement: interface is so flexible that parts can be interchanged or it is easy to write your own tool not to take care of proper initialization if you want to implement a molecular force field

—NOVELTY: Do we have one?

—We believe that BiochemicalAlgorithms.jl will be very helpful for scientist who want to do structural bioinformatics and don't know much about....additionally: visualization

—BiochemicalAlgorithms.jl already provides basic functionalities and interfaces for molecular mechanics but much is still to be done e.g., implementation of a docking interface, implementation of other force fields... we still have to do a lot of stuff to be feature-complete to BALL

—

6. References

[1] BALL Project Contributors. Ball project tutorial. `https://github.com/BALL-Project/ball/blob/master/doc/TUTORIAL/`, 2024. Accessed: 2024-12-16.

[2] Milan Bouchet-Valat and Bogumił Kamiński. Dataframes.jl: Flexible and fast tabular data in julia. *Journal of Statistical Software*, 107(4):1–32, 2023. doi:10.18637/jss.v107.i04.

[3] Milan Bouchet-Valat and Jacob Quinn. Tables.jl: A table interface for everyone. *Journal of Open Source Software*, 3(25):776, 2018. doi:10.21105/joss.00776.

[4] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments. *arXiv e-prints*, Aug 2016. 1608.04295.

[5] 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.

[6] Stefan Doerr, Matthew J. Harvey, Frank Noé, and Gianni De Fabritiis. Htmd: High-throughput molecular dynamics for molecular discovery. *Journal of Chemical Theory and Computation*, 12(4):1845–1852, 2016. doi:10.1021/acs.jctc.6b00049.

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

[8] 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.

[9] julia-vscode. Testitemrunner.jl: Run julia test items, 2022. Julia package.

[10] JuliaCI. Pkgbenchmark.jl: Benchmarking tools for julia packages, 2024. Julia package.

[11] Khronos Group. Opengl - the industry standard for high performance graphics. `https://www.opengl.org/`, 2024. Accessed: 2024-12-16.

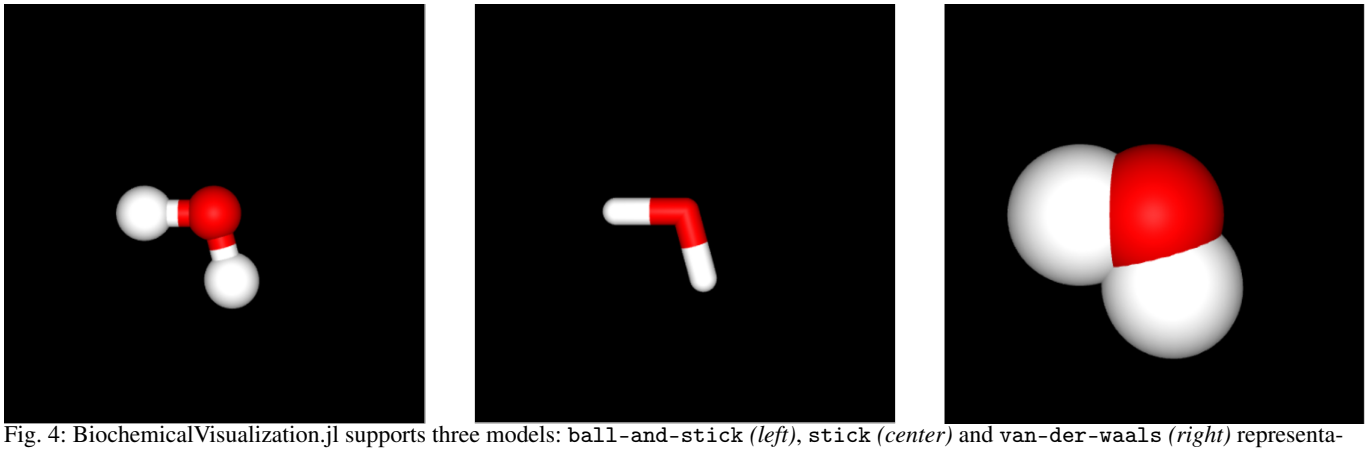

Fig. 4: BiochemicalVisualization.jl supports three models: `ball-and-stick` *(left)*, `stick` *(center)* and `van-der-waals` *(right)* representation of the water molecule as generated by the code listing 1.

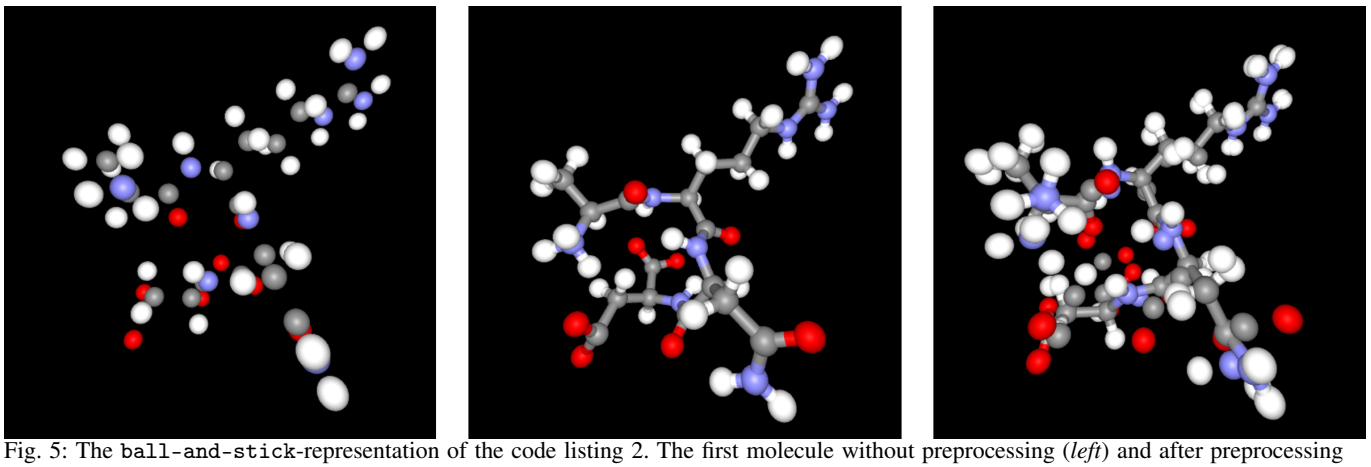

Fig. 5: The `ball-and-stick`-representation of the code listing 2. The first molecule without preprocessing (*left*) and after preprocessing (*center*). Finally, the two structures are superposed (*right*).

[12] 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.

[13] BioJulia Organization. Biojulia: Julia packages for bioinformatics and computational biology. `https://github.com/biojulia`, n.d.

[14] JuliaMolSim Organization. Juliamolsim: Molecular simulation in julia. `https://github.com/JuliaMolSim`, n.d.

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

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