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

BiochemicalAlgorithms.jl is framework for developing structure-based bioinformatics applications within the Julia ecosystem. Our library serves as a foundation providing rich functionality including file I/O, molecular modeling, molecular mechanics methods, and an accompanying visualization tool. BiochemicalAlgorithms.jl is based on Biochemical Algorithms Library (BALL), the largest open-source C++-framework of its kind. Our redesign emphasizes three design goals: ease of use, rapid application development (RAD), and functionality. Transitioning from C++ to Julia significantly simplified the realization of our design.

Keywords

Julia, Structure-based bioinformatics, Rapid Application Development (RAD), C++, BALL

1. Introduction

The aim of structure-based bioinformatics is the analysis and targeted manipulation of three-dimensional structures of biological macromolecules such as proteins and nucleic acids. This research field integrates disciplines ranging from fundamental physical laws to complex biochemistry knowledge and advanced numerical computing methodologies. For example, in molecular mechanics, a molecular force field is used to compute the energy of a structure. Structure-based bioinformatics encompasses applications such as molecular modelling, molecular dynamics (MD) simulations, and molecular docking. Molecular modelling techniques, particularly docking suites, attracted widespread interest during the COVID-19 pandemic: In the early phase, the protein structures were predicted based on sequence data as experimentally verified structures were not yet available. These structures were examined, including the analysis of the effects of the mutations originating from various virus strains. The knowledge of these molecular functions was then used in the context of rational drug design to find potential vaccines or drug therapeutics [15].
The COVID-19 pandemic highlighted the importance of these applications. However, most open-source software packages were developed much earlier, between 1995 and 2010. For instance, the molecular docking tools *AutoDock Vina* and its predecessor *AutoDock4* regained considerable popularity during the pandemic; however, they were developed much earlier in 2009 and 2010 [27, 16].
In the last decade before the pandemic, there have been no significant innovations in structure-based software developments. Several reasons contribute to the decreasing interest in this field. A crucial aspect is the availability of molecular structure data. Historically, the number of experimentally resolved structures was limited for many years [2] leading to a slowdown of the progress in this area. This changed in 2018 when DeepMind entered the CASP competition with AlphaFold [25] and, hence, put structure prediction in the spotlight again. Additionally, the number of experimentally determined structures with high resolution has increased dramatically through advances in cryo-electron microscopy in recent years. Nowadays, with the rapid rise of computed structures, availability no longer restrains the development of structural bioinformatics applications [28].

Software development in this field has been – and still is – typically challenging due to its interdisciplinary nature. The need for both numerical stability and computational efficiency, along with ease of use, has been a significant obstacle to RAD in open-source projects. Schroedinger is a closed-source framework providing packages for different molecular applications: `ProteinPreparationWizard` deals with the preprocessing of protein structural models and `LiveDesign` focuses on docking and designing ligands [23, 22]. These tools have the undeniable disadvantage of being closed-source and not free of charge.
For many open-source software packages, it is not uncommon to focus on implementing one specific task or algorithm (e.g., the introduction of a docking algorithm). The drawback of this approach is, that the user has to virtually glue several tools together providing the specific functionality. For instance, the structures have to be properly preprocessed before they can be used as input for a docking algorithm but these tools usually lack functionalities for preprocessing.
An exception to this single-purpose approach is the introduction of Biochemical Algorithms Library (BALL) by Kohlbacher *et al.* in 1996. BALL is a well-designed framework for molecular structure analysis written in C++. It offers file import and export, structure preprocessing, molecular mechanics, advanced solvation methods, and visualization options. Because of many contributions at the time BALL used to have one of the biggest user communities in this field. In 2010, a new version introduced Python bindings for enhanced usability.

A package for molecular dynamics simulation was published in more recent times and, similar to BALL it was written in C++ and included additional Python bindings [6]. While C++ is a natural choice to achieve the required efficiency of programs, it effectively hinders the rapid prototyping of molecular algorithms.

Developing software for structural bioinformatics is still challenging; however, choosing the programming language is not. Julia offers both the efficiency and numerical stability needed for molecular simulations along with rapid development capabilities. Several packages already exist in Julia related to structural bioinformatics – most prominently under two Github communities *Molecular Simulation in Julia* and *BioJulia* [18, 17]. The latter offers software packages for general biology approaches such as *BioSymbols.jl* for the representation of nucleic and amino acid primitive types as well as packages related to sequential bioinformatics. Most notably, it provides *BioStructures.jl* for reading and writing of macromolecular structures [9]. Additionally, Greener *et al.* published *Molly.jl*, a package for molecular simulations, which is part of the *Molecular Simulation in Julia* Github community [8]. Another interesting approach is *ProtoSyn.jl*; although it provides functionalities for analyzing peptides, its main focus is restricted to peptide design and engineering [21].
The mentioned Julia packages are limited to specific tasks e.g., *BioStructures.jl* is an excellent package for reading and writing PDB files. However, there remains a need for a platform that acts as an entry point by offering functionality encompassing an entire molecular structure pipeline.
We present BiochemicalAlgorithms.jl as a general-purpose framework for structure-based bioinformatics. BiochemicalAlgorithms.jl is a redesign of BALL and provides the foundation for molecular modelling and molecular simulation studies. Currently, we provide functionalities for:

—reading common data formats such as PDB, PDBx/mmCIF, HyperChem HIN, SDF (Structured Data File), and PubChem JSON

—preprocessing the input by preparing the entire system (e.g., adding missing hydrogens, bond computation, reconstruction of missing atoms, . . .)

—molecular mechanics through AMBER force fields

—mapping of structures

—structure minimization

—visualization of structures with BiochemicalVisualization.jl

In addition, BiochemicalAlgorithms.jl's intuitive interfaces enable users to develop their custom applications like the implementation of force fields for specific needs.

This article is organized as follows: First, we give a short background on BALL because this C++ framework motivated our design. In the next section, we depict how the switch from C++ to Julia improved our development. Thereafter, BiochemicalAlgorithms.jl's core is described. The ease of use and functionality are showcased in the application section. It contains four use cases including a comparison of C++ and Julia as well as visualizations.

2. BALL– Biochemical Algorithms Library

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 BALL's success lies in its

[1] An in-depth description of the entire BALL framework is beyond the scope of this article. Confer the main publications [14, 10] for more details.

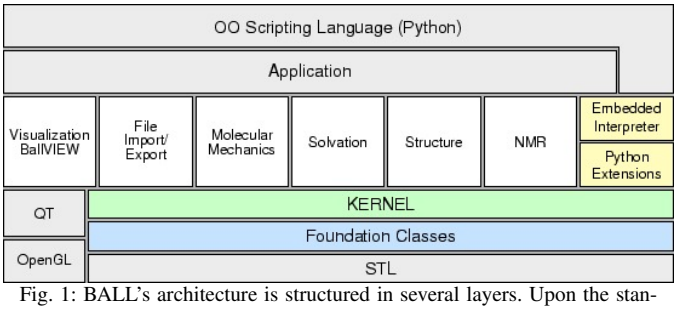

Fig. 1: BALL's architecture is structured in several layers. Upon the standard libary layer are the foundation classes and ontop of them the `KERNEL`. Several module extend the interface for visualization, file import and expoert, molecular mechanics, solvation, structure and NMR. The C++ written frameworkis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

sophisticated design; it employs an object-oriented approach with four design goals: *ease of use*, *robustness*, *openness*, and *functionality* The object-oriented approach facilitates ease of use in combination with the well-documented and intuitive interfaces. As can be seen in Figure 1, BALL's architecture is structured in several layers:

—The standard template library (STL) forms its base.

—The foundation classes provide general data structures such as hash sets and mathematical objects (e.g., matrices, vectors,...).

—The core consists of the `KERNEL` classes that contain data structures representing molecular entities.

—The basic components represent fundamental functionalities placed atop of this core layer; exceptions include the visualization module that is based on Qt and Open GL [26, 13].

—Finally, the application layers can be used to develop custom applications or leverage existing tools.

Hildebrandt *et al.* published an updated version in 2010, featuring Python bindings alongside installation with the use of CMake as build system – enhancements that improved usability and openness while allowing easier integration with external packages across different compilers/ operating systems [10].

BALL's uniqueness stems from its rich functionality integrated in a single easily extensible open-source platform. Figure 2 illustrates how `KERNEL` classes form three frameworks: the general molecular framework, the protein framework, the nucleic acid framework – all implemented through composite patterns [7]. 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.

These frameworks form the basis for the functionalities of the basic components ranging from preprocessing tools (file import/-export) to complex analyses (e.g., energy minimization/mapping) alongside advanced solvation methods. BALL is a well-tested library ensuring robustness of the provided functionalities.

BALL's robustness and well-designed structure has contributed significantly to its popularity – BALL used to have one of the biggest user communities in this field.

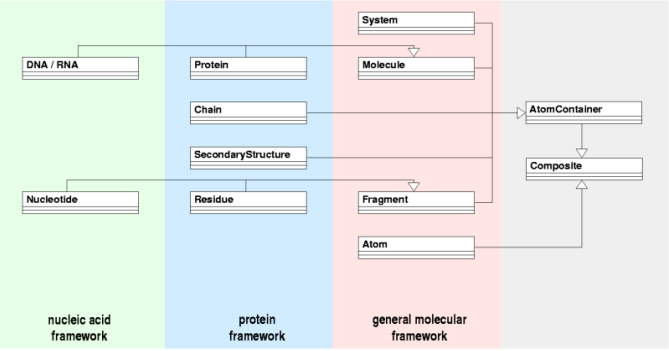

Fig. 2: UML class diagram of the `KERNEL` classes. The `KERNEL` classes form three frameworks and are implemented using the composite pattern[7]. The figure was taken from the official BALL documentation [1].

3. BiochemicalAlgorithms.jl

In this work we sought to redesign the popular BALL package for molecular analysis and simulation. This section examines reasons behind this redesign followed by descriptions detailing BiochemicalAlgorithms.jl's core implementations.

3.1 Reasons for a redesign: *why Julia?*

There remains an ongoing need in the structural bioinformatics ecosystem in Julia for a framework offering rich functionalities akin those provided by BALL. While the design goals are still valid, their realization in BALL is not contemporary anymore. The choice regarding programming language impacts implementation of the mentioned design goals massively. From today's perspective in particular with regard to its purpose as a platform for 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 library. For instance, significant time investment may often be required merely installing libraries alongside their dependencies. Despite using the CMake build system since version 1.3, setting up BALL remains a highly non-trivial task.
Moreover, knowledge surrounding utilized programming languages heavily influences development times. Low-level languages like C++ necessitate greater learning curves compared to scripting languages such as Python [19]. 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. In the latter, the core functionality is often implemented in a low-level programming language, ensuring performance, whereas higher-level programming languages facilitate user-friendly interfaces towards the core functionalities [5]. Julia was explicitly developed addressing these challenges [24].
Nevertheless, it is important to keep in mind that back in 1996 and still in 2010, C++ was the best choice for the implementation ensuring performance required specifically within the contexts involving molecular mechanics applications.

Switching our development from C++ to Julia has greatly simplified conforming the design goals:

—Ease of use: BiochemicalAlgorithms.jl's source code provides a better readability as the boilerplate code is massively reduced compared to BALL and the usage of C++. 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.

—Openness: Just like installations, the integration process surrounding external tools to BiochemicalAlgorithms.jl is straightforward. Our well-documented interfaces allow nearly seamless integration of custom applications.

—Robustness: One of the strengths 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 cases for the core structures as well as for the functionalities ensuring non-faulty behavior using `TestItemRunner.jl`[11]. Benchmark test cases are implemented in order to assess performance of typical tasks with the help of `BenchmarkTools.jl` and `Pkgbenchmark.jl` [4, 12].

—Functionality: BiochemicalAlgorithms.jl implements standard data structures for molecular entities and already provides different functionalities. These includes import of structures stored in common molecular data formats including PDB, PDBx/mmCIF, HyperChem HIN, SDF (Structured Data File), and PubChem JSON files. Molecular mechanics are offered through an interface for force fields and a concrete implementation for an AMBER force field. BiochemicalAlgorithms.jl provides algorithms for structure minimization and structure mappings. The interfaces are designed in a way that facilitates adoption, e.g., the implementation of custom force fields or changing an optimizer for the structure minimization.

3.2 The core representation

The core representation in BiochemicalAlgorithms.jl centers around the `System` data structure, which serves as the foundation for all applications within the framework. As shown in Figure 3, the system contains data structures for the representation of atoms, bonds, molecules, chains, residues, nucleotides, and fragments. These components are either generated explicitly or populated by reading input files (see code listings 1 and 2 in the applications section 4).

The atom representation with its position, velocity, and force contributes substantially to the framework's efficiency. After initially considering *DataFrames.jl* [3], we opted for a custom implementation of the *Tables.jl* interface [20]. The custom implementation enabled greater flexibility regarding the interface design and improved performance in initial benchmarks.
The custom implementation maintains compatibility with DataFrames.jl through the shared Tables.jl interface, allowing straightforward conversion when needed. Additionally, support for conversion to *AtomsBase.jl* representation is under development. Table 1 presents a performance comparison between BALL and BiochemicalAlgorithms.jl for processing an input structure with 892 atoms.
BiochemicalAlgorithms.jl is on par with its C++ predecessor in most tasks, with some operations like `compute_forces` being slower. It is important to keep in mind, that several years of development have led to a highly optimized code base in BALL, while our implementation still undergoes improvements.

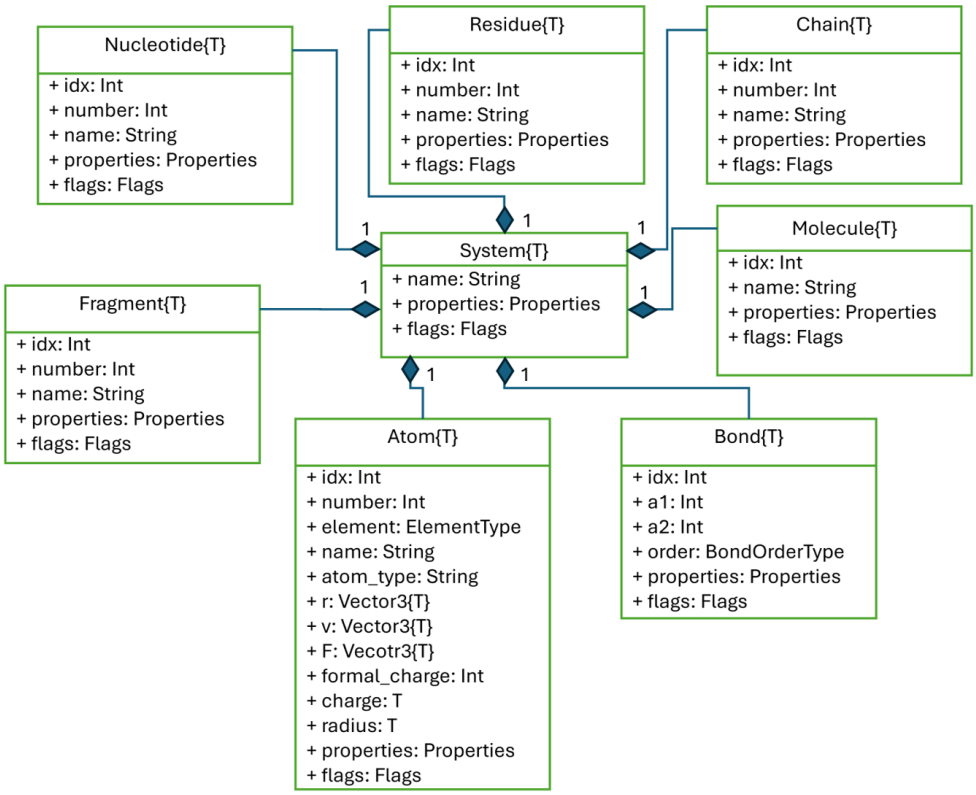

Fig. 3: UML diagram of the core of BiochemicalAlgorithms.jl. In the center resides the `System` interface. All other functionalities are grouped around that core piece. Only the most important functionalities of each class are shown.

Table 1. : Comparison of time requirements of the Amber force field implementations. The input file consisted of 892 atoms and was processed on the same system (AMP EPYC 7713 CPU, S/C/T=2/64/1).

Description	BALL	BiochemicalAlgorithms.jl
`compute_energy`	3.89 ms	4.706 ms
`compute_forces`	3.56 ms	33.590 ms
`update`	34.41 ms	89.635 ms
`setup`	72.04 ms	70.604 ms

4. Applications

In this section, we chose four use cases to illustrate BiochemicalAlgorithms.jl's capabilities in terms of functionality and usability, from basic operations to more advanced applications, while highlighting the advantages of using Julia for bioinformatics tasks. We begin with a simple example to show the creation and usage of core structures. Next, we compare two different configurations of the same molecule. Finally, we want to demonstrate the elegance of Julia code in comparison to C++ in the context of BALL and BiochemicalAlgorithms.jlİn the last application, we briefly introduce the accompanying visualization tool BiochemicalVisualization.jlthat has been developed alongside the BiochemicalAlgorithms.jl framework.

4.1 Generating a water molecule

The core of BiochemicalAlgorithms.jl is represented in a class diagram (Figure 3), which illustrates the framework's intuitive design and straightforward component interactions as can be seen in code listing 1.

Code 1: Intuitive usage of BiochemicalAlgorithms.jl core components

```
using BiochemicalAlgorithms
using BiochemicalVisualization

sys = System()
h2o = Molecule(sys)

o1 = Atom(h2o, 1, Elements.O, radius = 1.40f0)
h1 = Atom(h2o, 2, Elements.H, radius = 1.10f0)
h2 = Atom(h2o, 3, Elements.H, radius = 1.10f0)

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)
```

We have carefully chosen intuitive names for classes representing molecular entities and related functionalities. The `System` class

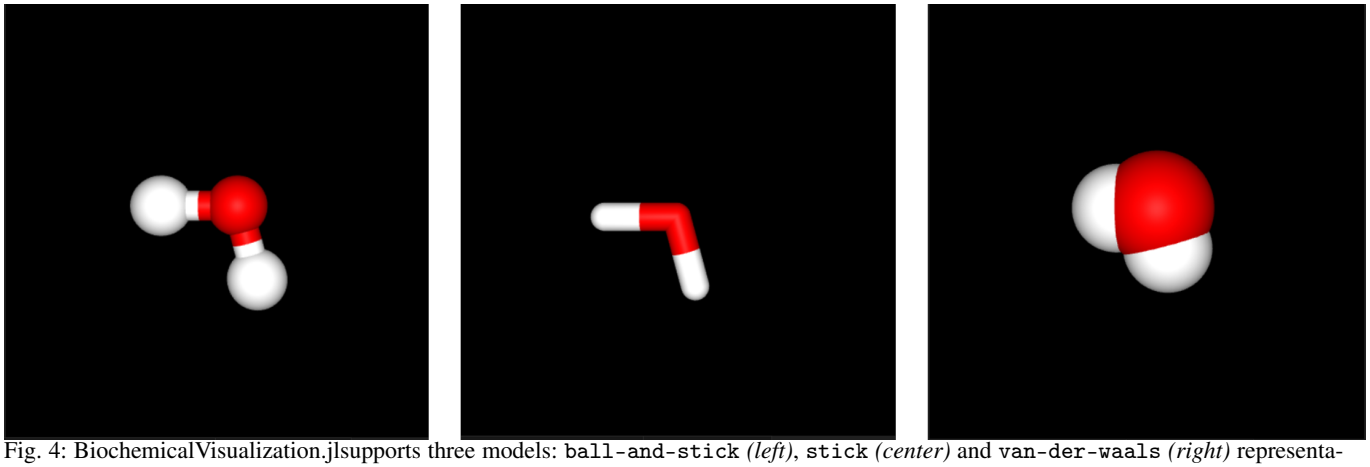

Fig. 4: BiochemicalVisualization.jlsupports three models: `ball-and-stick` *(left)*, `stick` *(center)* and `van-der-waals` *(right)* representation of the water molecule as generated by the code listing 1.

serves as the central element of any application. If not explicitly created, a default system is generated automatically. Atoms can be created along with their corresponding bonds and will automatically be incorporated in the defined system. The resulting system, containing a water molecule in this example, can be visualized using the BiochemicalVisualization.jl tool (Figure 4). More details about the visualization capabilities are provided in the subsequent section of the paper.
This approach emphasizes ease of use and functionality, two of the key design goals mentioned earlier for BiochemicalAlgorithms.jl. The intuitive interface and automatic system generation contribute to RAD, another stated goal of the framework.

4.2 RMSD computation and Application of AMBER force field

This example demonstrates the use of BiochemicalAlgorithms.jlfor a common task in structural analysis: comparing two (or more) molecular structures. The process involves several steps (see code listing 2):

—Loading structures: Two PDB files are loaded into a `Vector` of `System`, rather than a single system as in the previous example.

—Preprocessing: The systems are preprocessed using the functionalities provided by the `FragmentDB` interface, a database containing known fragments of molecules:
  —Normalizes different naming standards
  —Reconstructs missing parts of molecules
  —Creates bonds (as the PDB format often lacks complete bond information)

—Force field application: Each structure is applied to a molecular force field, specifically the Amber force field, and the energy of each system is computed.

—Structure mapping: The structures, which are different configurations of the same molecule, are mapped onto each other.

—RMSD computation: The Root Mean Square Deviation (RMSD) is calculated both before and after the mapping process.

This example showcases BiochemicalAlgorithms.jl's extensive functionality in just a few lines of code. The careful preparation steps taken for the systems are visually represented in Figure 5. The process demonstrates the framework's capability to handle complex structural analysis tasks efficiently, from file input and preprocessing to energy calculations and structure comparison, aligning with the design goals of functionality and ease of use.

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]))
```

4.3 RAD in BALL and BiochemicalAlgorithms.jl

RAD 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.
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 of the 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 is smaller than 6 Å.

The code listing 3 shows the solution for the task in C++. Due to readability, the necessary header files 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

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
```

in the next step.

Although the code is functional, it demonstrates the verbosity of C++ compared to the solution in Julia 4. 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.

It is important to note here that we did not actually call the functions. In Julia, only two additional lines of code are necessary: one for reading the structures used as input and one line for calling the function `count_contacts` with the input. The resulting snippet is then ready to be run from a Julia REPL without any further circumstances. In contrast, in the case of the C++-program we would have to write a main function, load the structures, call the function `count_contacts`. We would needed to include the necessary header files. Then the resulting code would have to be compiled and linked to the BALL framework. Even if we used the *CMake* build system, which makes it easier to link and generate an executable of our code to BALL, a *CMakeLists.txt* file is required to be written. Of course, in order to link to the BALL framework, it needs to be built ideally with the same compiler settings. For the purpose of building BALL a quite long lists of dependencies have to be built in advance for even just the core BALL functionalities. After managing all these steps successfully, we can finally call the executable to test the code snippet 3.

This example only serves to illustrate the basic approach for generating a small example using BALL or BiochemicalAlgorithms.jl.

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 image on the left represents the raw input read from the underlying PDB file, the image in the middle shows the same input after preprocessing it with the fragment data base (lines 4-7). Finally, the mapping of both structures is shown in the image on the right. 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.
The visualizations based on BiochemicalVisualization.jl can be integrated directly into Jupyter Notebooks or Visual Studio Code making the analysis even more convenient.

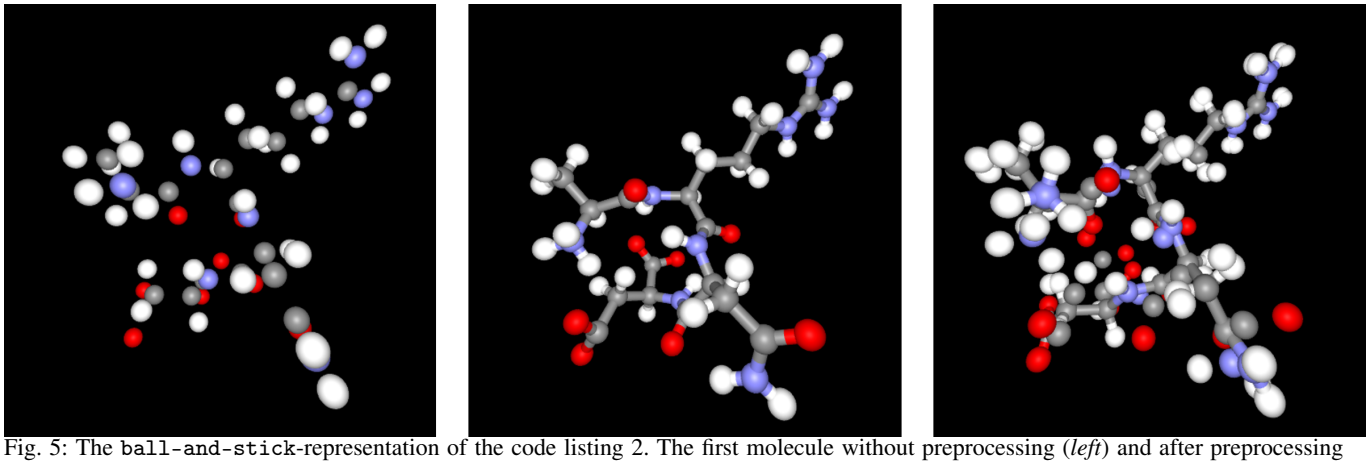

Fig. 5: The `ball-and-stick`-representation of the code listing 2. The first molecule without preprocessing (*left*) and after preprocessing (*center*). Finally, the two structures are superposed (*right*).

5. Conclusion

In this manuscript, we introduced BiochemicalAlgorithms.jl, a comprehensive framework designed for RAD in the field of structure-based bioinformatics. Unlike many existing Julia packages in this field that typically focus on single tasks, our library serves as a versatile foundation that encompasses a wide range of functionalities, including file I/O, molecular modelling, molecular mechanics methods, and is accompanied by a visualization tool.
Changing our development platform from C++ to Julia has greatly simplified conforming to the design goals: ease-of-use, openness, robustness, and functionality.

We believe that our framework facilitates both novice and experienced users in conducting molecular structure analysis with minimal effort. BiochemicalAlgorithms.jl provides a robust yet flexible core with additional functionalities. The integration of the fragment database is particularly valuable for structure preprocessing, including normalization of different naming standards, reconstruction of missing fragments, and bond computation. Additionally, the implemented visualization tool allows for immediate visual inspection of the structures.
Thereby, BiochemicalAlgorithms.jl is not intended to replace functionality already provided by packages inside the Julia ecosystem such as *BioStructures.jl*, but rather to provide a general platform allowing interoperability of functionalities. BiochemicalAlgorithms.jl includes well-defined interfaces, such as those for molecular force fields, empowering experienced users to implement custom applications. Table 1 shows timings for our implementation of an AMBER force field in comparison to BALL. These results indicate comparable performance with some tasks being slower in BiochemicalAlgorithms.jlẆe are committed to improving performance of the force field implementation and working on alternative force field implementations such as CHARMM, which will be available in the near future.

Future directions will include an extensive benchmark study to evaluate our implementation against its predecessor, BALL. This will involve creating a modern benchmarking suite in C++ for BALL to allow comparison with results from *BenchmarkTools*.
Overall, we see BiochemicalAlgorithms.jl as a valuable contribution to the field of structural bioinformatics in Julia, combining ease of use with powerful functionality to support a wide range of applications.

Acknowledgements

Parts of this research were conducted using the supercomputer MOGON NHR and/or advisory services offered by Johannes Gutenberg University Mainz (hpc.uni-mainz.de), which is a member of the AHRP (Alliance for High Performance Computing in Rhineland Palatinate, www.ahrp.info) and the Gauss Alliance e.V. The authors gratefully acknowledge the computing time granted on the supercomputer MOGON NHR at Johannes Gutenberg University Mainz (hpc.uni-mainz.de).

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] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic acids research*, 28(1):235–242, 2000.

[3] 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.

[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] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, Massachusetts, 1994.

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

[9] Joe G Greener, Joel Selvaraj, and Ben J Ward. Biostructures.jl: read, write and manipulate macromolecular structures in julia. *Bioinformatics*, 36(14):4206–4207, 2020. doi:10.1093/bioinformatics/btaa502.

[10] 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.

[11] julia-vscode. Testitemrunner.jl: Run julia test items. `https://github.com/julia-vscode/TestItemRunner.jl`, 2022. Julia package.

[12] JuliaCI. Pkgbenchmark.jl: Benchmarking tools for julia packages. `https://github.com/JuliaCI/PkgBenchmark.jl`, 2024. Julia package.

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

[14] 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.

[15] Nitin Kumar, Abhinav Jain, Ashwani Kumar, Pankaj Kumar, Anurag Mishra, Kumardeep Chaudhary, and Gajendra P.S. Raghava. Structural bioinformatics resources for sars-cov-2. *Briefings in Bioinformatics*, 22(3):931–943, 2021. doi:10.1093/bib/bbaa261.

[16] Garrett M. Morris, Ruth Huey, William Lindstrom, Michel F. Sanner, Richard K. Belew, David S. Goodsell, and Arthur J. Olson. Autodock4 and autodocktools4: Automated docking with selective receptor flexibility. *Journal of Computational Chemistry*, 30(16):2785–2791, 2009. doi:10.1002/jcc.21256.

[17] BioJulia Organization. Biojulia: Julia packages for bioinformatics and computational biology. `https://github.com/biojulia`. Accessed: December 13, 2024.

[18] JuliaMolSim Organization. Juliamolsim: Molecular simulation in julia. `https://github.com/JuliaMolSim`. Accessed: December 13, 2024.

[19] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998. doi:10.1109/2.660187.

[20] Jacob Quinn, Bogumił Kamiński, David Anthoff, Milan Bouchet-Valat, Tamas K. Papp, Takafumi Arakaki, Rafael Schouten, Nick Robinson, mathieu17g, Okon Samuel, Jarrett Revels, ExpandingMan, Eric Hanson, PhD Anthony Blaom, Alex Arslan, Jerry Ling, Jiahao Chen, Josh Day, José Bayoán Santiago Calderón, Julia TagBot, Kristoffer Carlsson, Lilith Orion Hafner, Miles Cranmer, Randy Zwitch, Steven G. Johnson, Tom Gillam, Yue Yang, spaette, strickek, and Jacob Adenbaum. JuliaData/Tables.jl: v1.12.0, July 2024. doi:10.5281/zenodo.12753139.

[21] Diogo Santos-Martins and Sérgio F. Sousa. Protosyn.jl. https://github.com/sergio-santos-group/ProtoSyn.jl, 2023. Julia package.

[22] Inc. Schrödinger. Livedesign: Your complete digital molecular design and discovery lab. `https://www.schrodinger.com/platform/products/livedesign/`. Accessed: 2024-12-17.

[23] Inc. Schrödinger. Protein preparation wizard. `https://www.schrodinger.com/life-science/learn/white-papers/protein-preparation-wizard/`. Accessed: December 17, 2024.

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

[25] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander W R Nelson, Alex Bridgland, et al. Improved protein structure prediction using potentials from deep learning. *Nature*, 577(7792):706–710, 2020.

[26] The Qt Company. Qt 5. `https://doc.qt.io/qt-5/`, 2024. Accessed: 2024-12-16.

[27] Oliver Trott and Arthur J. Olson. Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of Computational Chemistry*, 31(2):455–461, 2010. doi:10.1002/jcc.21334.

[28] Mihaly Varadi, Samuel Anyango, Mandar Deshpande, Jeyan Nair, Felicity Natassia, Gergana Yordanova, Deming Yuan, Oana Stroe, Amos Pajon, Jake McGreig, Quentin Szabo, Tom Kosciolek, Daniel Lowe, Michele Magrane, Andrew P. Harrison, M. S. Madhusudhan, Peer Bork, Bence Sipos, David Suveges, Silvio C. E. Tosatto, Christine A. Orengo, Christoph Steinbeck, Jiannis Kourelis, Andrew R. Leach, Gerard J. Kleywegt, Sameer Velankar, Julian Gough, Óscar Conchillo-Solé, Alex Bateman, Christine A. Orengo, Sameer Velankar, and Julian Gough. Alphafold protein structure database: Updates, enhancements, and future directions. *Nucleic Acids Research*, 51(D1):D439–D446, 2023. doi:10.1093/nar/gkac1234.