juliacon

ProtoSyn.jl: a novel platform for computational molecular manipulation and simulation with a focus on protein design

José M. S. Pereira[1], José R. B. Gomes[1], and Sérgio M. Santos[1]

[1]Dep. Química & CICECO, Universidade de Aveiro, Aveiro, Portugal

ABSTRACT

Modern protein design and drug discovery workflows harness the potential of computer simulations to prototype new effectors from the desired task, with applications in therapeutics, environment remediation and industrial settings, among others. In the recent decades, the field of computer sciences has seen tremendous growth, not only with the constant improvement in hardware power and availability, but also with the expansion of new techniques such as the usage of artificial intelligence, cloud & high-performance computing, GPU acceleration and the introduction of modern programming languages. However, open-source scientific software targeted towards molecular simulation and protein design has lagged behind, mainly relying in outdated and poorly documented software, written in low-level programming languages and with decades of patches and hotfixes. ProtoSyn.jl, a novel platform for computational molecular manipulation and simulation, aims to offer an improved user experience. Written in Julia, a modern high-level scripting language, ProtoSyn.jl is fast, modular and includes a complete and comprehensive documentation manual with examples and use cases. Harnessing the power of open-source software, ProtoSyn.jl has the potential to modernize the way the scientific community uses simulation tools, bridging the gap between computational and experimental scientists. ProtoSyn.jl version 1.0, released July 2021, can be accessed at https://github.com/sergio-santos-group/ProtoSyn.jl

Keywords

Julia, Protein Design, Molecular manipulation, Simulation

1. Introduction

Traditionally, the goal of protein design and drug discovery was mainly pursued via experimental directed evolution or blind search [1, 11]. Such processes are lengthy and expensive, and often yield poor results. With the advent of modern computers, simulations and *in silico* prototyping took precedence, quickly gaining traction as powerful tools to lower development times and costs, while increasing the accuracy of experimental efforts [9]. In fact, over the last 20 years, the number of published papers in the computational chemistry field increased over 5-fold (measured by searching for the "computational chemistry" keywork in ScienceDirect).

On the scope of protein design (and protein folding prediction), computer aided approaches have steadily increased in both complexity and accuracy, in great part due to the hegemony of the Rosetta software [16]. In 1999, Simons *et al.*, from the Baker group, presented the first examples of homology-based methods for protein folding prediction, resulting in the release of the first version of Rosetta, written in Fortran [21]. This early version of Rosetta was used to publish Top7, the first protein design of a novel topology [12]. This software was later translated into C++, in 2006, and wrapped multiple times in other languages, exposing some of the core functionalities, such as in Python, with Pyrosetta, in 2009 [4]. Over the last decade, the Rosetta family of scripts and software programs grew to include a large number of applications and scientific scopes (from ab initio modelling to NMR structure prediction) [13]. Recently, and following incredible the work of AlphaFold, in CASP 13 [20], and AlphaFold2 in CASP 14 [22, 10], the trRosetta application was published, using neural networks to predict protein structures [24].

Other software solutions of notice include PyMOL and USCF ChimeraX [8], albeit with a much narrower field of application (mainly focusing of molecular manipulation and visualization).

It is possible to observe a common trend in the last 20 years of scientific software development for protein design: small scripts or software packages are developed and published for a specific task, often using Rosetta as the backend, and later bundled together. More often than not, computational design efforts make use of multiple tools, in multiple programming languages. This paradigm offers blatant disadvantages, such as the need for file type parsing scripts, non-existent or improper documentation, performance loss due to data sharing via the file system and the incapacitation of most users to add code modifications due to the usage of legacy or low-level programming languages (such as Fortran or C++, respectively). This, unsurprisingly, offer a massive resistance to adoption of unified workflows and posts a large entry barrier to non-specialized members of the scientific community [18]. As such, and besides commonly known best practices and recommendations for writing good open source software [19] (for example, using a code hosting and version control, namely GitHub), four additional requirements for a modern software solution tailored for protein design and molecular simulation can be defined:

—Performace – Protein design efforts often rely on algorithms with millions of calculation steps. For this reason, even a slight slowdown in performance can cost hours or days in simulation time. Languages such as Python and R are notoriously slow 17, while lower-level languages such as C or C++ offer much better performance at the expense of readability and an out-of-the-box experience. This is known as the "two-language problem" and arises

from the need to have the core code in a low-level language and user interfaces in a higher-level language. A modern solution should avoid the two-language problem while maintaining the highest possible performance.

—Modular – A modular solution allows versatility without increasing code entropy: users should be able to add new functionalities as modules, making use of existing structures, in the same programming language and following the same general guidelines. This removes the need for data sharing via the file system and usage of file type parsing scripts, while allowing greater flexibility in customization of protocols and linkage between different algorithms.

—Well documented – Application-specific scripts are often developed and maintained "in-house", frequently causing proper documentation to be overlooked. Code without appropriate documentation is often the root cause for costly and time-consuming errors, elevating the entry barrier to new users (especially non-specialized scientists). Good documentation should also include a properly maintained and curated list of examples and tutorials, as well as adequate unit testing, allowing for easier implementation of new features without the risk of code breaking changes.

—Scalable - In the past decade, a healthy number of new technologies have gained traction in the scope of computer sciences, such as the usage of machine learning and artificial intelligence models, GPU computing and cloud computing. A modern tool for protein design should be scalable in order to incorporate new emerging technologies as they become available in order to boost accuracy, security and performance.

Despite the undeniable success of Rosetta, with its many iterations and years of development, the current patchwork of scripts and applications does not fully comply to these requirements: the Rosetta (C++) and PyRosetta (Python) still fall into the "two-language problem"; the Rosetta software is, for the most part, interfaced with by multiple individual applications, each for a particular use (although, as an alternative, the RosettaScripts allow users to create and modify protocols using an XML-based syntax, exposing some of the core functionality provided by Rosetta [6]); the PyRosetta documentation is infamous for its lack of documentation and outdated examples; and finally, Rosetta does not directly benefit from modern GPU acceleration platforms. These shortcomings have been identified before, with interesting works such as the EGAD Library, but the proposed solutions failed to satisfy all of the above requirements, preventing mass adoption and eventually falling into discontinuity [5].

In conclusion: there's room for improvement.

2. Developing a new tool

ProtoSyn.jl (whose name is a mesh between the words "Prototyping" and "Synthetic", and whose attempt at a logo is shown in Figure 1) aims to be a novel platform for molecular manipulation and simulation, with an emphasis on protein design.
The development of ProtoSyn.jl started in early 2018, fueled by the need for a more open, well documented and flexible tool for protein design. Since the project's inception, three key concepts

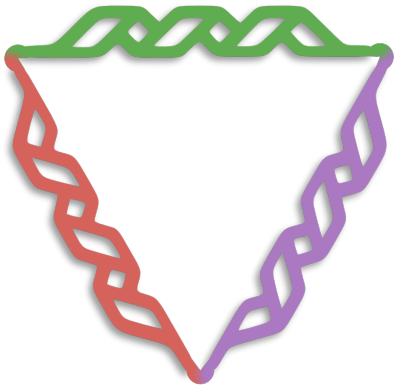

Fig. 1. **ProtoSyn.jl's logo.** Attempts to represent 3 α-helix structures using Julia's logo colors.

guided the development process: ProtoSyn.jl should be an open-access project, built on modularity and aimed for non-expert usage. ProtoSyn.jl is completely written in Julia [3, 2], a modern programming language specifically developed to solve the two-language problem. Besides taking advantage of the momentum being felt by the Julia environment, with a growing and active community, ProtoSyn.jl also benefits from many of the inherent features of this programming language, such as an easy scripting syntax ("Looks like Python . . . ") and good performance (". . . runs like C!"), access to low-level methods (such as GPU and SIMD acceleration, using CUDA.jl and SIMD.jl), direct access to Python and C code calls (using PyCall.jl), native support for distributed computing and a growing number of useful packages (such as DataFrames.jl, Plots.jl and emerging solutions in the molecular simulation environment, such as Molly.jl and AtomsBase.jl). As such, ProtoSyn.jl's current flagship features include:

—Complete molecular manipulation tools – One of the main objectives of ProtoSyn.jl is to offer a playground experience, where users can freely modify peptides and proteins. Sometimes, a hard part of the protocol is to simply introduce a quick change in the structure/sequence of the peptide. With this goal in mind, ProtoSyn.jl makes available methods in order to remove, add and mutate residues, remove and add sidechains, saturate structures with hydrogens, cut, copy and paste whole molecules or fragmented selections of chains, explore and create new rotamer libraries, freely rotate dihedrals, apply secondary structures, etc. Using ProtoSyn.jl's Builder module adds the possibility to create and append peptides from scratch, using only the amino acids sequence, and adds the capacity to include non-canonical aminoacids (NCAA) and post-translational modifications (PTMs) such as methylation and phosphorylation.

—Complex simulation algorithms – In order to test new designs, ProtoSyn's simulation environment uses Monte Carlo, iterated local search (ILS) and steepest descent algorithms, among others, to explore the conformational space. Besides the existing methods, users can also define custom drivers for the simulation.

—Fully customizable energy functions – As with any simulation package, a simulation's accuracy is dictated by the energy function employed. Besides the default energy function provided by ProtoSyn.jl, a rich library of potentials & machine learning mod-

els is made available for users to pick and choose from, creating a custom energy functions specifically suited to the task at hands. Some examples of available energy function terms include the usage of contact maps, the TorchANI machine learning potential, geometrical hydrogen bonds, steric clashes, coarse-grained solvation energy, Generalized-Born (GB) solvation potential, etc. Besides the existing components, users are encouraged to prototype and test new energy function components, with a complete guide in the documentation and auxiliary methods made available. Developing energy function components is inherently hard, thus providing a centralized space to carry out benchmarks, parameterization efforts and "plug-and-play" testing has been a major focus during development of ProtoSyn.jl.

—Rich selection syntax – An ample list of selection methods can be employed and combined with ease, using logical operators, to specify with precision the target atoms or regions for a particular modification, simulation or energy measurement.

—GPU and cloud computing – Making use of Julia's inherent characteristics, ProtoSyn.jl includes native support for GPU acceleration of most heavy calculations. Including the Distributed.jl package extends the usage of ProtoSyn to multiple threads (and even multiple machines), further accelerating the simulation of multiple replicas simultaneously. This paradigm can also be applied in dedicated HPC contexts, easily incorporating ProtoSyn.jl in commonly used cloud computing platforms.

In short: ProtoSyn intends to be a playground for prototyping new designs and ideas, enhancing the early stages of new protocols and projects by quickly deploying fast, scalable and accurate simulations of the mutation, interaction of modification of interest.

3. Under the hood

Following the established practice, the main data structure in ProtoSyn.jl is called a Pose. In ProtoSyn.jl, this is a complete description of a molecular system, incorporating both a directed Graph (this defines the nature of the particles in the system, as well as the interactions between each other) and a State (defining the position of the particles and the energy landscape of the system), as schematically described in Figure 2.
Firstly, the directed Graph introduces the hierarchical organization of a system: Atoms are grouped in Residues, which in turn are organized in Segments, which, finally, collectively form a Topology. Secondly, the directed Graph, as the name implies, introduces the concept of parenthood between particles in the system: each atom has a parent, and may have one or more child atoms. The same concept is extended to Residues. Ultimately, the initial atom in a Pose is a child of the Root structure, a group of three virtual atoms.

This organization becomes particularly interesting when coupled with the State hybrid coordinate system: each system of atoms is described both by the cartesian coordinates (where each position is defined by the X, Y and Z coordinates in 3D space), as well as by the internal coordinates (in which case each position is defined by the distance, angle and dihedral angle to the parent/ascendents particles). Both coordinate systems are interchangeable and synched during simulation: in certain methods, such as a dihedral rotation,

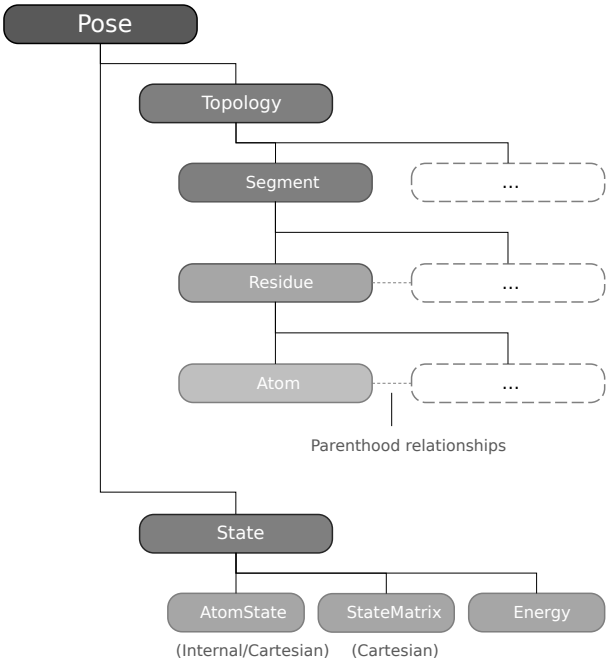

Fig. 2. **ProtoSyn.jl's main structure class, the Pose.** A Pose is described by both a directed graph (under the class Topology) and a State. The Topology holds meta information regarding the nature and relationships between particles in a system, being hierarchically comprised of Segment, Residue and Atom instances. Both Residue and Atom instances have parenthood relationships between themselves. The State holds information regarding the position of the atoms both in internal and cartesian coordinates, as well as the current system's energy.

internal coordinates are both easier to use and faster to calculate, while in others, such as a rigid body movement, cartesian coordinates take precedence. Besides the atomic positions, a Pose's State also holds information regarding the atomic partial charges (when necessary) and the current system's energy. Most of ProtoSyn.jl's molecular manipulation methods (such as any function implied in a mutation or in building a peptide from a sequence) make use of a stochastic L-system formal grammar [17]: the Residues in ProtoSyn.jl are included in a grammar and can be combined from strings by a collection of production rules and an initial axiom. In the case of peptides, the production rule is linear (the peptide bond). However, the usage of this paradigm opens the path to the incorporation of randomly generated ramified carbohydrates and glycoproteins (with α-1-4 and α-1-6 bonds, for example), as well as allowing the incorporation of new amino acids in the available grammar, even with multiple connections between Residue instances (as is the case in some non-canonical amino acids). Besides allowing the incorporation of newly defined NCAA instances by the user, the overall architecture of ProtoSyn.jl also allows for easy introduction of modifications to existing amino acids, such as in phosphorylation and methylation post-translational processes, a recent avenue of research that has gained traction in the last few years [7, 14].

Another interesting implication of employing L-grammars as the basis of ProtoSyn.jl is that, for most applications, atom types are

replaced by a template-based approach. For example, each amino acid template includes all the required information about each individual atom, such as the default partial charge, among others, instead of requiring a costly atom typing process. In other words, the atom typing problem is replaced by an atom naming problem: for correct retrieval of information from templates, it must be possible to have a one-to-one relationship between a structure and a template. In some cases, such as when working with proteins, atom names (in this case, in the context of amino acids) are extensively conventionalized, with specific nomenclatures accepted and widely employed. Therefore, this one-to-one relationship, via the correct atom name attribution, becomes quite easy and already established in most cases. The above-mentioned types and methods define most of the Core module of ProtoSyn.jl: any function defined here is agnostic to the molecule type being used. Taking advantage of the multiple dispatch paradigm employed by the Julia programming language [15], methods are extended in particular modules (such as the Peptides module), introducing specific functions that assume the molecules being simulated are of a given nature (for example, methods in the Peptides module might assume the input Pose has CA atoms). Each of these modules is usually subdivided in the following sub-modules, as schematically illustrated in Figure 3:

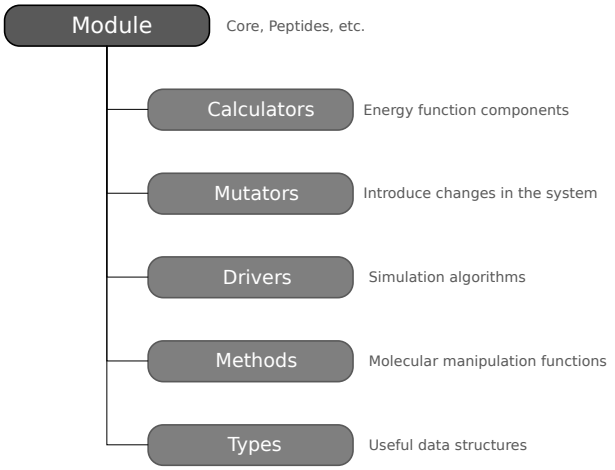

Fig. 3. **ProtoSyn.jl's modules organization.** ProtoSyn.jl's code is organized in multiple modules, based on the target's molecule type. The Core module is agnostic and supplies basic functions. Specialty modules (such as the Peptides module) define Calculators, Mutators and Drivers specific for a molecule type (proteins, in this example). Calculators are individual energy function components, used to evaluate a given Pose in accordance with a specified potential; Mutators introduce individual changes in a system; and Drivers are algorithms that drive the system from a state to another, in accordance with some rule. In addition, general molecular manipulation functions and useful data structures can also be found at the Methods and Types directories, respectively.

4. Conclusion

Even though ProtoSyn employs this seemingly simple architecture, given the modular nature of most components, arbitrarily complex protocols and simulations can be easily constructed. In one such example, ProtoSyn was employed to determine an approximated folding structure of a small peptide, PDB 2A3D [23]. This protocol is shared here as an illustration of the usage of ProtoSyn.jl: even the simplest of protein folding prediction problems pose a challenge to any molecular manipulation and simulation software, thus benchmarking the employed energy function and performance of the platform. For the full code (including comments), see https://github.com/JosePereiraUA/ProtoSyn-use-cases. As shown in Figures 4 and 5, ProtoSyn.jl is able to correctly simulate the folding patterns of a small peptide (down to Cα-Cα RMSD of 3.93 Å), plugging in several state-of-the-art models from the literature in an extremely easy way. Using the Zhang Lab TM-score online server, the predicted structure shows a TM-score of 0.5927 (excluding the sidechains). Overall, the distributed simulations, despite running for 1 million steps each, concluded in a reasonable amount of time, taking less than 120 hours to complete all 1000 replicas. As such, the distributed computing set up run an average of 2300 conformational samplings per second. Figure 6 showcases ProtoSyn.jl usage in high-resolution side chain clash solving.

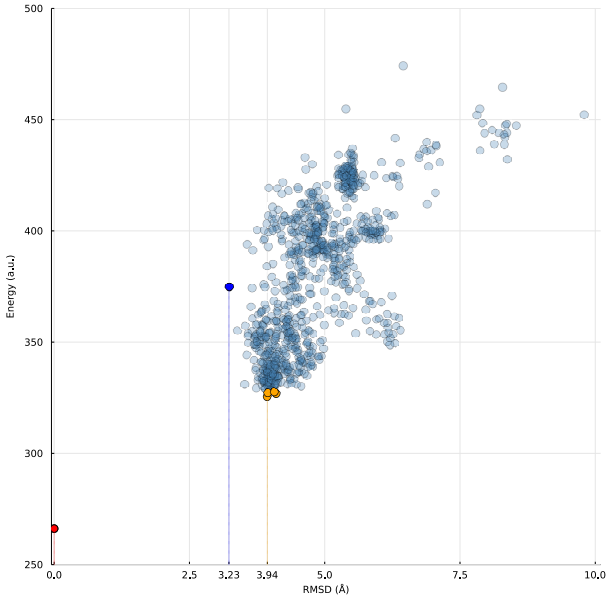

Fig. 4. **Low-resolution folding prediction using ProtoSyn.jl.** In light blue, all 1000 replicas show a good folding funnel. Highlighted in yellow, the 5 lowest energy structures. Within this set, the lowest energy structure (identified by the low-resolution algorithm) showcases 3.93 Å in RMSD. Marked in dark blue, the lowest RMSD structure (not identified by ProtoSyn.jl as a low-energy structure) shows a 3.23 Å RMSD. The base truth (i. e.: the crystallographic model) is shown in red.

ProtoSyn.jl's development constitutes a first attempt at a Julia-based molecular manipulation and simulation software. In conclusion, ProtoSyn.jl does not intend to re-invent the wheel: previous software solutions for molecular manipulation and simulation have proven, time and time again, their accuracy and performance, forever changing the landscape of how protein design is pursued. Technology development, however, is continuously adding new and better ways to do stuff and, as such, ProtoSyn.jl intends to introduce much needed quality-of-life improvements while lowering the entry barrier to non-expert users. Furthermore, it is the developer's

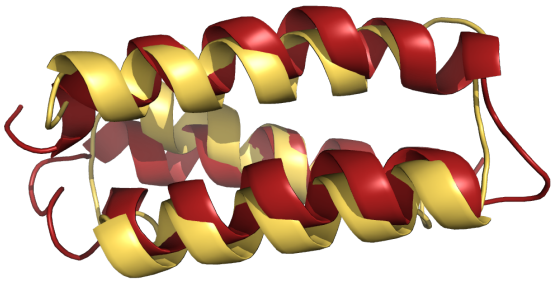

Fig. 5. **Lowest energy candidate.** In yellow, the lowest energy Pose from the low-resolution simulation shows a 3.93 Å RMSD value when compared with the crystalline structure of the 2A3D peptide, in red.

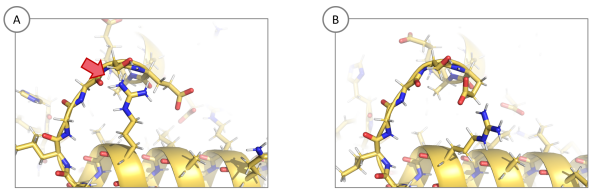

Fig. 6. **Solving sidechain clashes.** In yellow, the lowest energy Pose from the low-resolution simulation is minimized (in a high-resolution simulation) to solve atomic clashes (marked by a red arrow). A – The initial sidechain recovery is agnostic to its surroundings, often inducing atomic clashes when placing sidechains. B – After a few steps of a Monte-Carlo simulation, initial clashes are solved and the minimized structure shows a lower all-atom RMSD value.

view that a modern scientific software's potential is only truly unlocked by allowing and encouraging input from various sources and scientific backgrounds, in an open-source setting, while maintaining and curating a complete, correct and exhaustive documentation and examples suite. Given the implementation within the Julia environment, further interface development with emerging packages, such as Molly.jl, DTFK.jl and AtomsBase.jl will further strengthen the available suite of tools available for computational chemists interested in exploring protein design within the Julia ecosystem.

5. References

[1] Frances H. Arnold. When blind is better: Protein design by evolution. *Nature Biotechnology*, 16:617–618, 1998.

[2] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. Julia: dynamism and performance reconciled by design. *Proceedings of the ACM on Programming Languages*, 2:1–23, 10 2018.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59:65–98, 1 2017.

[4] Sidhartha Chaudhury, Sergey Lyskov, and Jeffrey J. Gray. Pyrosetta: A script-based interface for implementing molecular modeling algorithms using rosetta. *Bioinformatics*, 26:689–691, 1 2010.

[5] Arnab B. Chowdry, Kimberly A. Reynolds, Melinda S. Hanes, Mark Voorhies, Navin Pokala, and Tracy M. Handel. An object-oriented library for computational protein design. *Journal of Computational Chemistry*, 28:2378–2388, 11 2007.

[6] Sarel J. Fleishman, Andrew Leaver-Fay, Jacob E. Corn, Eva Maria Strauch, Sagar D. Khare, Nobuyasu Koga, Justin Ashworth, Paul Murphy, Florian Richter, Gordon Lemmon, Jens Meiler, and David Baker. Rosettascripts: A scripting language interface to the rosetta macromolecular modeling suite. *PLoS ONE*, 6, 2011.

[7] Donghyeok Gang and Hee sung Park. Noncanonical amino acids in synthetic biosafety and post-translational modification studies. *ChemBioChem*, 22:460–468, 2 2021.

[8] Thomas D. Goddard, Conrad C. Huang, Elaine C. Meng, Eric F. Pettersen, Gregory S. Couch, John H. Morris, and Thomas E. Ferrin. Ucsf chimerax: Meeting modern challenges in visualization and analysis. *Protein Science*, 27:14–25, 1 2018.

[9] Po-Ssu Huang, Scott E. Boyken, and David Baker. The coming of age of de novo protein design. *Nature*, 537:320–327, 9 2016.

[10] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A.A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature 2021 596:7873*, 596:583–589, 7 2021.

[11] Christian Jäckel, Peter Kast, and Donald Hilvert. Protein design by directed evolution. *Annual Review of Biophysics*, 37:153–173, 5 2008.

[12] Brian Kuhlman, Gautam Dantas, Gregory C Ireton, Gabriele Varani, Barry L Stoddard, and David Baker. Design of a novel globular protein fold with atomic-level accuracy. *Science*, 302:1364–1368, 11 2003.

[13] Julia Koehler Leman, Brian D. Weitzner, Steven M. Lewis, Jared Adolf-Bryfogle, Nawsad Alam, Rebecca F. Alford, Melanie Aprahamian, David Baker, Kyle A. Barlow, Patrick Barth, Benjamin Basanta, Brian J. Bender, Kristin Blacklock, Jaume Bonet, Scott E. Boyken, Phil Bradley, Chris Bystroff, Patrick Conway, Seth Cooper, Bruno E. Correia, Brian Coventry, Rhiju Das, René M. De Jong, Frank DiMaio, Lorna Dsilva, Roland Dunbrack, Alexander S. Ford, Brandon Frenz, Darwin Y. Fu, Caleb Geniesse, Lukasz Goldschmidt, Ragul Gowthaman, Jeffrey J. Gray, Dominik Gront, Sharon Guffy, Scott Horowitz, Po Ssu Huang, Thomas Huber, Tim M. Jacobs, Jeliazko R. Jeliazkov, David K. Johnson, Kalli Kappel, John Karanicolas, Hamed Khakzad, Karen R. Khar, Sagar D. Khare, Firas Khatib, Alisa Khramushin, Indigo C. King, Robert Kleffner, Brian Koepnick, Tanja Kortemme, Georg Kuenze, Brian Kuhlman, Daisuke Kuroda, Jason W. Labonte, Jason K. Lai, Gideon Lapidoth, Andrew Leaver-Fay, Steffen Lindert, Thomas Linsky, Nir London, Joseph H. Lubin, Sergey Lyskov, Jack Maguire, Lars Malmström, Enrique Marcos, Orly Marcu, Nicholas A. Marze, Jens

Meiler, Rocco Moretti, Vikram Khipple Mulligan, Santrupti Nerli, Christoffer Norn, Shane Ó'Conchúir, Noah Ollikainen, Sergey Ovchinnikov, Michael S. Pacella, Xingjie Pan, Hahnbeom Park, Ryan E. Pavlovicz, Manasi Pethe, Brian G. Pierce, Kala Bharath Pilla, Barak Raveh, P. Douglas Renfrew, Shourya S.Roy Burman, Aliza Rubenstein, Marion F. Sauer, Andreas Scheck, William Schief, Ora Schueler-Furman, Yuval Sedan, Alexander M. Sevy, Nikolaos G. Sgourakis, Lei Shi, Justin B. Siegel, Daniel Adriano Silva, Shannon Smith, Yifan Song, Amelie Stein, Maria Szegedy, Frank D. Teets, Summer B. Thyme, Ray Yu Ruei Wang, Andrew Watkins, Lior Zimmerman, and Richard Bonneau. Macromolecular modeling and design in rosetta: recent methods and frameworks. *Nature Methods*, 17:665–680, 7 2020.

[14] Ken Nagata, Arlo Randall, and Pierre Baldi. Incorporating post-translational modifications and unnatural amino acids into high-throughput modeling of protein structures. *Bioinformatics*, 30:1681–1689, 6 2014.

[15] Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: a rational reconstruction. *Proceedings of the ACM on Programming Languages*, 2:1–27, 10 2018.

[16] José M. Pereira, Maria Vieira, and Sérgio M. Santos. Step-by-step design of proteins for small molecule interaction: A review on recent milestones. *Protein Science*, 30:1502–1520, 8 2021.

[17] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. Graphical modeling using l-systems, 1990.

[18] Ilan Samish. Achievements and challenges in computational protein design. *Methods in Molecular Biology*, 1529:21–94, 2017.

[19] Qusay Idrees Sarhan. Best practices and recommendations for writing good software. *The Journal of the University of Duhok*, 22:90–105, 11 2019.

[20] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander W.R. Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David T Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 577:706–710, 2020.

[21] K. T. Simons, R. Bonneau, and David Baker. Ab initio protein structure prediction of casp iii targets using rosetta. *Proteins*, Suppl. 3:171–176, 1999.

[22] Kathryn Tunyasuvunakool, Jonas Adler, Zachary Wu, Tim Green, Michal Zielinski, Augustin Žídek, Alex Bridgland, Andrew Cowie, Clemens Meyer, Agata Laydon, Sameer Velankar, Gerard J. Kleywegt, Alex Bateman, Richard Evans, Alexander Pritzel, Michael Figurnov, Olaf Ronneberger, Russ Bates, Simon A.A. Kohl, Anna Potapenko, Andrew J. Ballard, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Ellen Clancy, David Reiman, Stig Petersen, Andrew W. Senior, Koray Kavukcuoglu, Ewan Birney, Pushmeet Kohli, John Jumper, and Demis Hassabis. Highly accurate protein structure prediction for the human proteome. *Nature 2021 596:7873*, 596:590–596, 7 2021.

[23] Scott T.R. Walsh, Hong Cheng, James W. Bryson, Heinrich Roder, and William F. Degrado. Solution structure and dynamics of a de novo designed three-helix bundle protein. *Proceedings of the National Academy of Sciences of the United States of America*, 96:5486–5491, 5 1999.

[24] Jianyi Yang, Ivan Anishchenko, Hahnbeom Park, Zhenling Peng, Sergey Ovchinnikov, and David Baker. Improved protein structure prediction using predicted interresidue orientations. *Proceedings of the National Academy of Sciences of the United States of America*, 117:1496–1503, 11 2020.