gmxapi: a GROMACS-native Python interface for molecular dynamics with ensemble and plugin support

M. Eric Irrgang[1], Caroline Davis[1], and Peter M. Kasson[1,2*]
[1]Departments of Molecular Physiology and Biomedical Engineering, University of Virginia, Charlottesville VA 22908
[2]Science for Life Laboratory, Department of Cell and Molecular Biology, Uppsala University, 75124 Uppsala, Sweden
[*] To whom correspondence should be addressed: kasson@virginia.edu, Box 800886 Charlottesville VA 22908. ORCID 0000-0002-3111-8103.

Abstract

Gmxapi provides an integrated, native Python API for both standard and advanced molecular dynamics simulations in GROMACS. The Python interface permits multiple levels of integration with the core GROMACS libraries, and legacy support is provided via an interface that mimics the command-line syntax, so that all GROMACS commands are fully available. Gmxapi has been officially supported since the GROMACS 2019 release and is enabled by default in current versions of the software. Beyond simply wrapping GROMACS library operations, the API permits several advanced operations that are not feasible using the prior command-line interface. First, the API allows custom user plugin code within the molecular dynamics force calculations, so users can execute custom algorithms without modifying the GROMACS source. Second, the Python interface allows tasks to be dynamically defined, so high-level algorithms for molecular dynamics simulation and analysis can be coordinated with loop and conditional operations. Gmxapi makes GROMACS more accessible to custom Python scripting while also providing support for high-level data-flow simulation algorithms that were previously feasible only in external packages.

Introduction

As molecular dynamics simulations have become more complex and mature as scientific tools, typical simulation use is shifting from manual invocation of a few simulations and analysis tools to pre-defined simulation and analysis protocols, often involving many simulation trajectories. In addition, custom applications for advanced sampling[1-4] or molecular structure refinement[5-7] are becoming more common, where the molecular dynamics engine is used as part of a more complex data integration protocol. In these more complex use scenarios, toolchain and file-system management as well as execution of many simulations can become limiting factors. As a result, many scientists either develop custom scripts, custom modifications of the simulation source code, or adapt general-purpose workflow engines[8-10] to molecular simulation tasks. There also exist some molecular-simulation-specific workflow engines, frequently coupled to the underlying molecular simulation code via command-line interfaces[11-16]. All of the above can be brittle, particularly without robust APIs for simulation interfaces to workflow engines. Integration of parallel analysis into unified jobs can also be challenging whether this is performed via built-in tool parallelism or user-level coding, such as Python multiprocessing packages or workflow/execution managers[9,10,12,13,17-19].

We previously reported on gmxapi 0.0.4[20], which allows Python driven molecular dynamics in GROMACS[21,22] to be extended at run time with custom researcher code. Here, we describe features present in gmxapi 0.2 and beyond, which offers both more advanced ensemble simulation logic and a feature-complete interface for GROMACS tools and analysis. This framework allows ensemble methods to be implemented with less code, and without patching an official GROMACS release. Data-flow-oriented programming logic and general paradigm for integrating new software tools further enhance the utility of the package. Both the libgmxapi C++ interface and the gmxapi Python package are now maintained and distributed with GROMACS; gmxapi 0.2 is integrated with GROMAC 2021.

Design and Implementation

We used pybind11[23] to implement Python bindings to a C++ API for the GROMACS[21,22] molecular simulation library. The design concept of gmxapi, as with many modern high-level interfaces, is for the user to construct a computational graph where execution is then managed by lower layers of the software stack. Similar to Keras[24], we recognize that explicit data-flow programming is not natural for all users and thus provide an interface where data-flow can be specified explicitly or implicitly. Figure 1 illustrates the gmxapi paradigm in which control signals and data are treated in the same framework. Branching and other control events require that some work is "dynamic", such that the complete workflow graph may not be determined until runtime. However, the workflow can be fully specified in terms of workflow commands and data primitives (Figure 2).

Data flow formalism

Data-flow formalisms have become increasingly popular in contemporary workflow engines due to their ability to separate work specification and task execution, permitting more straightforward resolution of dependencies and optimization of computation and data movement. For the same reasons, execution in gmxapi is deferred as much as possible to when and where it is required. The high-level gmxapi Python interface creates and executes a directed acyclic graph (DAG) specifying the computational work to be done. Nodes of the work graph represent discrete *operations* that produce and consume well defined inputs and outputs. Operation and data references in the work graph are proxy objects for the computational tasks and data until the graph is run. Execution management may be optimized by running only as much of the graph as necessary to satisfy explicit data dependencies. In addition, in order to minimize unnecessary data movement, most gmxapi operations do not transfer data from the C++ library to the Python interpreter unless explicitly requested by the user. When a gmxapi command is used to add work to the work graph, it returns a reference to the graph node representing the operation. The operation's output attributes may be used as inputs to further commands, and may be used as "Future" results, as discussed below. Calling "result()" on a Future forces dependency resolution and data localization.

Python interface

The gmxapi Python interface consists of five categories of operations: 1) typing and logical operations, 2) bound GROMACS API calls, 3) legacy GROMACS command-line operations, 4) utilities for user creation of new gmxapi operations, and 5) looping and conditional operations. These are described in sequence below.

The simplest gmxapi operations manipulate signals, data topology, or typing. “join_arrays”, “logical_not”, and “make_constant” have strict definitions for inputs and outputs (Figure 2a). These operations are also composable: “concatenate_lists” is a helper function that reduces a number of inputs in terms of “join_arrays” (Figure 2b).

The preferred mechanism for interfacing to GROMACS is via bound API calls. Commands like “read_tpr”, “modify_input”, and “mdrun” use a binary Python extension module written in C++ using pybind11 to interact with GROMACS operations and data through libgmxapi, which is a C++ library installed by default with recent versions of GROMACS.

In order to provide legacy support for the full range of GROMACS command-line operations, “gmxapi.commandline.cli()” is a simple pure-Python gmxapi operation that wraps command-line tools. It is a thin wrapper of the Python “subprocess” module, so command inputs and outputs are embedded in the argument list parameter. Instances of “gmxapi.commandline.cli” are not by themselves conducive to DAG representations of data flow, so a helper function creates additional gmxapi primitives to provide a consistent set of named inputs and outputs. “gmxapi.commandline_operation()” generates a graph of gmxapi primitives around “cli” to translate input and output arguments into operation inputs and outputs, as specified by the user (Figure 2b).

In order to provide extensibility, arbitrary user code can be transformed into a gmxapi operation by decorating a Python function definition with “@function_wrapper” (Figure 2c). As an example, “commandline_operation” is written in this manner.

Looping and conditional commands are critical to adaptive ensemble simulation[25] and thus form a key part of the gmxapi repertoire. Conditional iteration takes the form of conventional gmxapi command syntax but relies on some metaprogramming under the hood. The “while_loop” command allows a graph to be dynamically extended as a chain of repeated operations, including fused operations constructed with the “subgraph” tool. A subgraph allows inputs and outputs to be expressed in terms of other gmxapi operations. When used with “while_loop”, internal subgraph state can be propagated from one iteration to the next to provide a consistent environment similar to a standard loop.

The work graph

The work graph permits data flow topology to be represented independently from execution strategies or run time resource assignment details. An array of input sources provided to a command generates a corresponding set of tasks, such as for trajectory ensemble simulations. Because such ensembles may be coupled (and gmxapi currently does not have a mechanism to specify that an ensemble is uncoupled), arrays of

simulations must currently be co-executed through the gmxapi mpi4py executor. Operations defined with "@function_wrapper", including the other built-in operations, are assumed to be uncoupled, and are launched in a sequence determined by the DAG topology during recursive resolution of data dependencies, executed sequentially by the simple built-in gmxapi 0.1 executor.

The work graph enables straightforward dependency resolution: commands are specified to operate on handles to work rather than requiring the concrete work object, so chains of simulations can be expressed before the first simulation begins. A command is then ready to execute once its inputs have been fully resolved. Because operations and data flow are represented as a directed acyclic graph (DAG), arbitrarily complex topologies can be expressed unambiguously without unexpected side-effects. Trajectories can be forked or extended without re-executing simulation segments or overwriting previous results.

Execution and control flow

The gmxapi interface enables several levels of control logic, described below. Native Python logic (if/else statements, for loops, etc.) can be used with gmxapi operations. However, since gmxapi operations construct a work graph that is then executed, Python conditionals operate on the work graph construction rather than control flow in work graph execution. Native gmxapi logic such as "while_loop" provides a more efficient way to express adaptivity within a sequence of gmxapi operations. Work-graph execution and high-level control can be enforced in Python-level logic using the result() method, which causes explicit execution of all code required to produce the requested result. In contrast, gmxapi-level logic operates on Futures and is thus deferred until graph execution.

Low-level control logic is available within runtime libraries and is designed for use by plugin code. For instance, gmxapi registers with the GROMACS "StopSignal" facility to enable plugin code to stop a simulation based on external criteria, such as when a statistical estimator has converged. Additional examples of low-level operations include a "ReduceAll" callable that collects data across an in-flight ensemble. This operation can be used for adaptive updates such as modifying biasing forces in restrained-ensemble simulations[26,27] or to update estimators and determine when a simulation ensemble should be terminated.

Gmxapi as a means to extend MD code without source modification

Gmxapi establishes a facility[28] for providing MD extension code objects to the GROMACS simulator during launch. Binary objects can be loaded at runtime via the Python interpreter. This allows custom code to be executed during the MD integration loop with minimal overhead. Python facilitates plugin binding, but once the simulation launches, GROMACS and the extension code communicate directly with C++ interfaces, so there is no overhead added by Python. Since this only requires a standard GROMACS installation with gmxapi, it permits flexible extension of the molecular dynamics code without modifying the main codebase or libraries.

Gmxapi does not prescribe a C/C++ bindings strategy. We used pybind11 for GROMACS bindings, but the protocols for Python interaction with compiled code rely only on the Python C API (PyCapsule) and the public libgmxapi C++ interface.

Results

Typical molecular dynamics workflows use a chain of tools to prepare simulation input, including command-line programs where input and output can only be expressed as file paths. The syntax of gmxapi syntax permits the output of any tool to be used as the input to another with consistent Python syntax. Figure 3 illustrates a chain of GROMACS tools preparing simulation input, feeding into "gmxapi.mdrun" to run a batch of simulations.

To illustrate the application of gmxapi, we demonstrate refinement of an HIV gp41 conformational ensemble based on DEER spectroscopy data. We illustrate this using a starting crystal structure for the BG505 SOSIP of HIV gp41[29] and previously reported DEER spectroscopy data of 5 different spin labels positions on this SOSIP[30]. The gp41 trimer was asymmetrically restrained (each of the three monomer-monomer distances was sampled separately), and the previously reported BRER algorithm for refining heterogeneous conformational ensembles[28] was applied. This involves a custom force plugin for GROMACS, and gmxapi was used to run an array of 250 refinement simulations, each randomly sampling target distance values from the experimental distribution for each measured spin-label pair. Further details are given in the Supplement. Full convergence of the refinement would require additional simulation sampling of the 15-dimensional experimental distribution, but even after 24 wall-clock hours of simulation a reasonable sampling of the experimental distributions was obtained (Figure 3). This application illustrates the ease of simulation ensemble management and application of custom biasing algorithms within GROMACS facilitated by gmxapi.

Availability and Future Directions

The libgmxapi C++ interface was released as part of GROMACS 2019 release and has been part of a standard GROMACS installation since 2020. Gmxapi 0.2 (described here) shipped with GROMACS 2021. Continued development of gmxapi will expand the number of bound GROMACS API calls and reduce the need for legacy command-line support as well as generalize the plugin interface. At the programming level, one key future direction is exactly that—an increased use of the “Future” paradigm in Python. Python design patterns using Futures are becoming more widespread and standardized [14,31] as a way to refer to results that have not yet been calculated. Some additional refinement of the gmxapi Future protocol is needed to be fully compatible with native and third-party frameworks for concurrent or asynchronous program flow.

Code and data availability are specified below:
The gmxapi Python package is also maintained as part of the GROACS repository at https://gitlab.com/gromacs/gromacs. It can be installed from the GROMACS source (https://gitlab.com/gromacs/gromacs/-/tree/master/python_packaging/src) or from https://pypi.org/project/gmxapi/ with “pip”, but needs to be told which GROMACS installation to use. The documentation at https://manual.gromacs.org/current/gmxapi

provides details. During installation, the gmxapi Python package builds a C++ extension module against the GROMACS installation.

Custom molecular dynamics extension code is illustrated in a “sample_restraint” package: https://gitlab.com/gromacs/gromacs/-/tree/master/python_packaging/sample_restraint. Bias-resampling ensemble refinement (BRER) MD restraint potentials[28] are forked from this sample code and can be found at https://github.com/kassonlab/brer_plugin. Scripted BRER workflows are available at https://github.com/kassonlab/run_brer. The “run_brer” scripts were developed prior to gmxapi 0.1 and use internal logic to manage some data flow and to maintain workflow state. Both “run_brer” and “gmxapi” are being updated to illustrate that data flow and workflow state can be managed by the framework to allow simpler, more robust application code for applying new methods like BRER. Input data for the BRER example shown have been deposited at doi:10.5281/zenodo.5122931.

Gmxapi issues are tracked with the label “gmxapi” at https://gitlab.com/gromacs/gromacs/-/issues. Code contributions follow the GROMACS contribution procedure (https://manual.gromacs.org/current/dev-manual/contribute.html). However, gmxapi is intended to allow for maximal extensibility without requiring modification to the sources.

A discussion forum is available at https://gromacs.bioexcel.eu/tag/gmxapi.

Acknowledgements. The authors thank Mark Abraham for many helpful discussions. This work was supported by a MolSSI Software Fellowship to M.E.I. and R01 GM115790 and OAC-1835780 to P.M.K. Computational resources on Frontera were provided under NSF LRAC MCB20006.

Figures

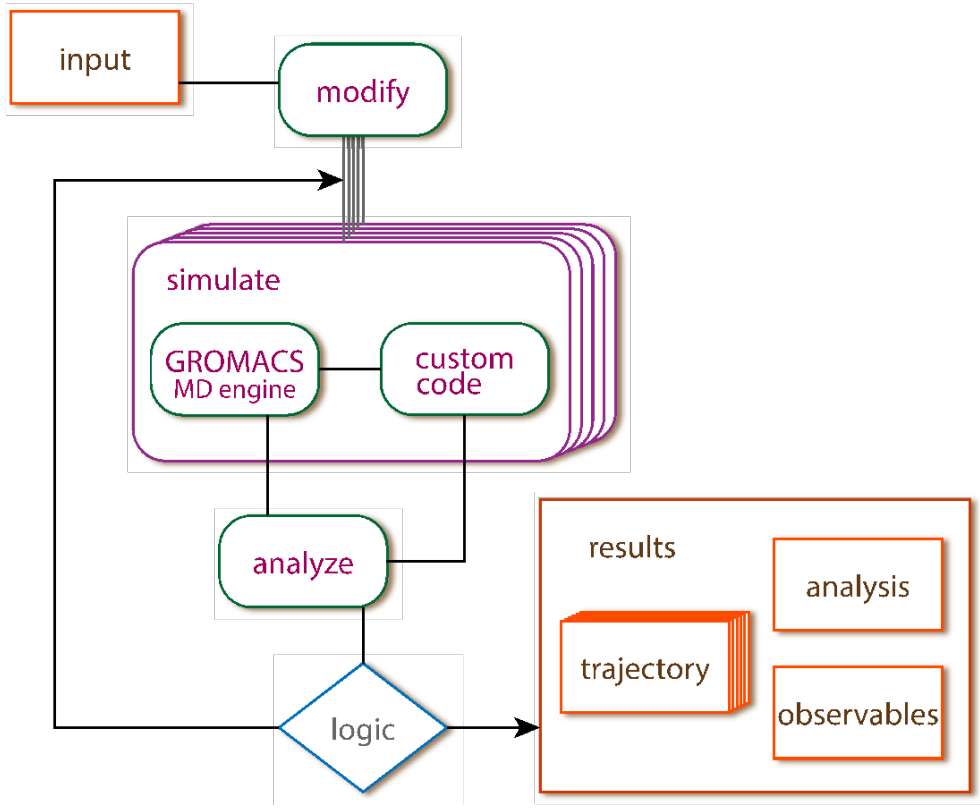

Figure 1: Schematic of data flow and control flow for a segment of a complex simulation workflow. Ensembles of simulations can be run (denoted by stacked rectangles) by gmxapi merely by passing an array of inputs instead of a single input. Custom plugins can interact with running MD simulations. Finally, conditional and looping logic can create high-level simulation algorithms.

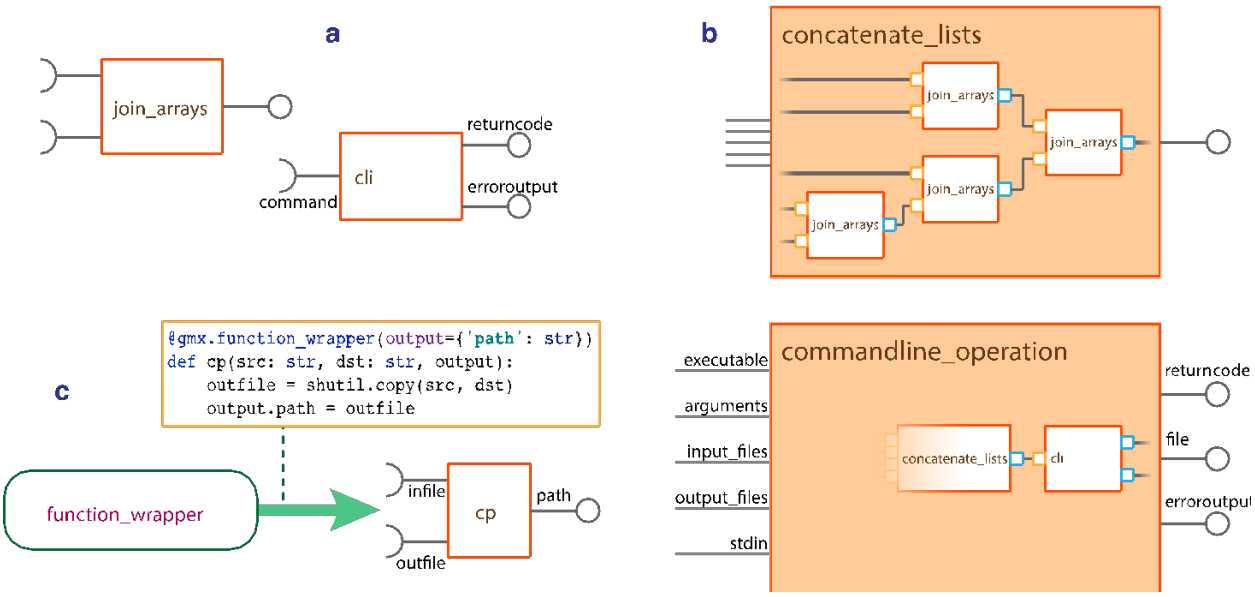

Figure 2: Schematics for gmxapi operations. As shown in panel (a), gmxapi operations have well defined inputs and outputs. These can operate on arrays of inputs and also include support for legacy GROMACS operations by wrapping the command-line toolset, as shown in panel (b). As shown in panel (c), the decorator @function_wrapper allows arbitrary user code to be transformed into a gmxapi operation.

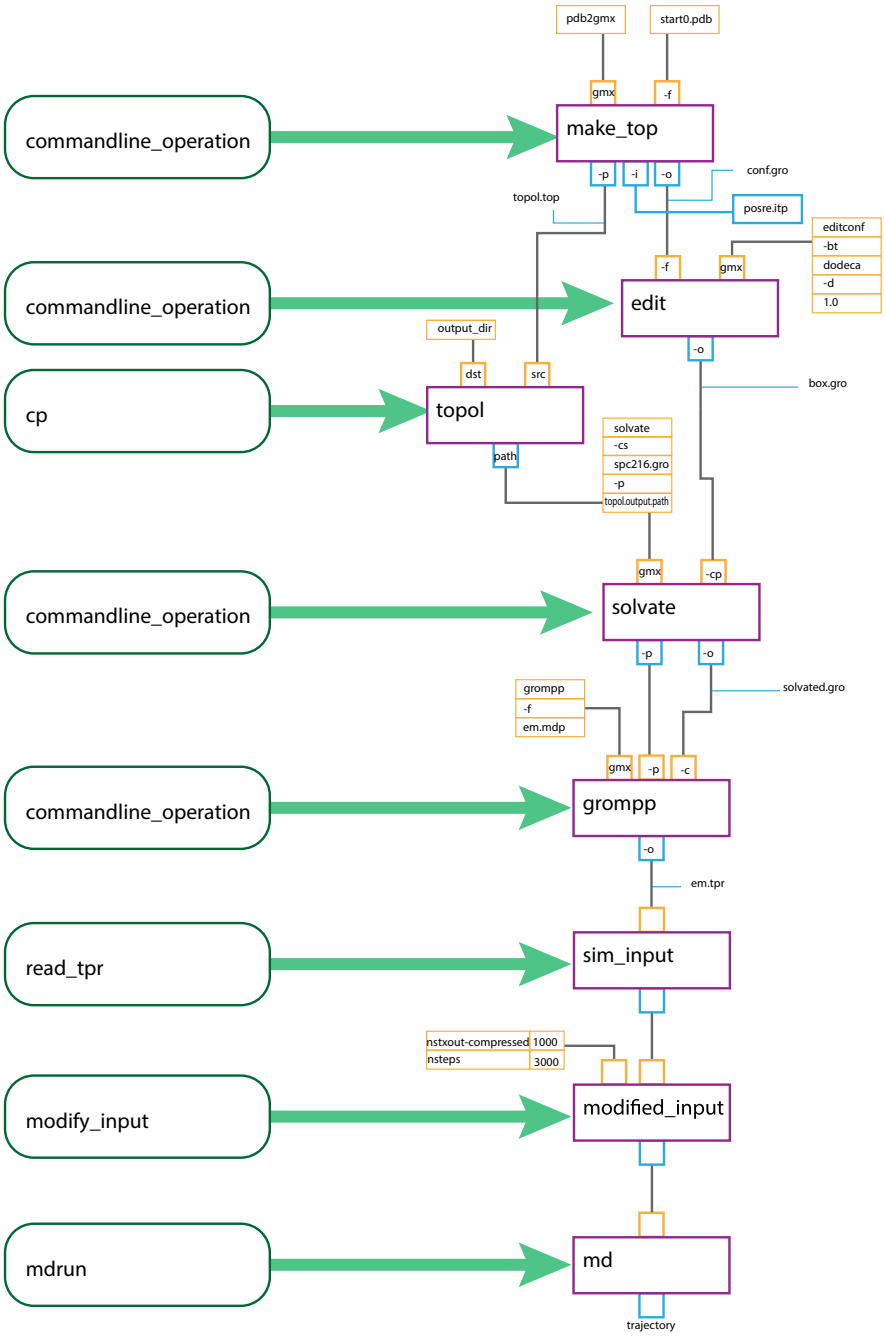

Figure 3: Detailed diagram of inputs, outputs, and operations in a chain of gmxapi operations. Operations are shown in green boxes, the corresponding nodes in the work graph in red boxes, inputs in orange boxes, and outputs in blue boxes. The diagram depicts a chain of GROMACS tools preparing simulation input, feeding into "gmxapi.mdrun" to run a batch of simulations.

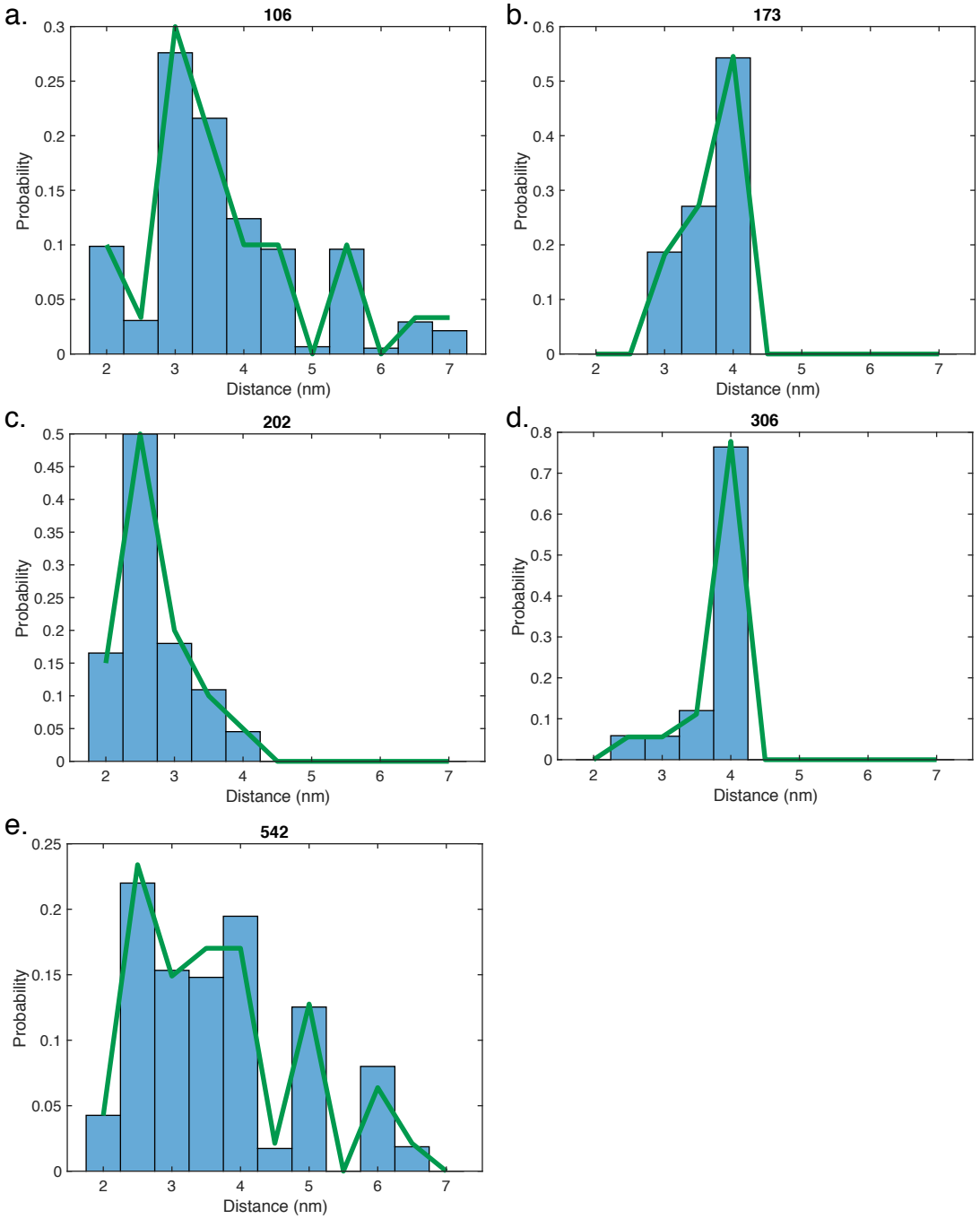

Figure 4: Residue-residue distance distributions in a simulated ensemble of HIV gp41. DEER spectroscopy was used to measure distance distributions between two monomers of the gp41 trimer. Thus, each residue designates a monomer-monomer residue pair. Panels a-e show plots for the 5 restrained residues in the HIV trimer: 106, 173, 202, 306, and 542 respectively. Discretized DEER distance distributions are plotted in green, and simulation results are plotted in blue bars. The simulation ensemble shows good convergence to the measured values.

Literature Cited

1. Bonomi M, Camilloni C (2017) Integrative structural and dynamical biology with PLUMED-ISDB. Bioinformatics 33: 3999-4000.
2. Tribello GA, Bonomi M, Branduardi D, Camilloni C, Bussi G (2014) PLUMED 2: New feathers for an old bird. Computer Physics Communications 185: 604-613.
3. Olsson S, Wu H, Paul F, Clementi C, Noe F (2017) Combining experimental and simulation data of molecular processes via augmented Markov models. Proc Natl Acad Sci U S A 114: 8265-8270.
4. Chodera JD, Noé F (2014) Markov state models of biomolecular conformational dynamics. Current opinion in structural biology 25: 135-144.
5. Heo L, Feig M (2018) Experimental accuracy in protein structure refinement via molecular dynamics simulations. Proceedings of the National Academy of Sciences 115: 13276-13281.
6. Perilla JR, Zhao G, Lu M, Ning J, Hou G, et al. (2017) CryoEM structure refinement by integrating NMR chemical shifts with molecular dynamics simulations. The Journal of Physical Chemistry B 121: 3853-3863.
7. Igaev M, Kutzner C, Bock LV, Vaiana AC, Grubmuller H (2019) Automated cryo-EM structure refinement using correlation-driven molecular dynamics. Elife 8.
8. Goecks J, Nekrutenko A, Taylor J, Team G (2010) Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. Genome Biology 11.
9. Deelman E, Singh G, Su M-H, Blythe J, Gil Y, et al. (2005) Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming 13: 219-237.
10. Wolstencroft K, Haines R, Fellows D, Williams A, Withers D, et al. (2013) The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. Nucleic Acids Research 41: W557-W561.
11. Pronk S, Larsson P, Pouya I, Bowman GR, Haque IS, et al. (2011) Copernicus: A new paradigm for parallel adaptive molecular dynamics. Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis: 60.
12. Pronk S, Pouya I, Lundborg M, Rotskoff G, Wesén B, et al. (2015) Molecular Simulation Workflows as Parallel Algorithms: The Execution Engine of Copernicus, a Distributed High-Performance Computing Platform. Journal of Chemical Theory and Computation.
13. Jain A, Ong SP, Chen W, Medasani B, Qu XH, et al. (2015) FireWorks: a dynamic workflow system designed for high-throughput applications. Concurrency and Computation-Practice & Experience 27: 5037-5059.
14. Babuji Y, Woodard A, Li Z, Katz DS, Clifford B, et al. Parsl: Pervasive parallel programming in python 2019. pp. 25-36.
15. Wilde M, Hategan M, Wozniak JM, Clifford B, Katz DS, et al. (2011) Swift: A language for distributed parallel scripting. Parallel Computing 37: 633-652.
16. Balasubramanian V, Bethune I, Shkurti A, Breitmoser E, Hruska E, et al. Extasy: Scalable and flexible coupling of md simulations and advanced sampling techniques; 2016. IEEE. pp. 361-370.
17. van der Aalst WMP, Jablonski S (2000) Dealing with workflow change: identification of issues and solutions. Computer Systems Science and Engineering 15: 267-276.
18. Ludascher B, Altintas I, Berkley C, Higgins D, Jaeger E, et al. (2006) Scientific workflow management and the Kepler system. Concurrency and Computation-Practice & Experience 18: 1039-1065.
19. Mattoso M, Dias J, Ocana KACS, Ogasawara E, Costa F, et al. (2015) Dynamic steering of HPC scientific workflows: A survey. Future Generation Computer Systems-the International Journal of Escience 46: 100-113.
20. Irrgang ME, Hays JM, Kasson PM (2018) gmxapi: a high-level interface for advanced control and extension of molecular dynamics simulations. Bioinformatics 34: 3945-3947.
21. Pronk S, Pall S, Schulz R, Larsson P, Bjelkmar P, et al. (2013) GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. Bioinformatics 29: 845-854.
22. Abraham MJ, Murtola T, Schulz R, Páll S, Smith JC, et al. (2015) GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. SoftwareX 1: 19-25.
23. Wenzel J, Rhinelander J, Moldovan D (2017) pybind11–Seamless operability between C++ 11 and Python.
24. Chollet F (2015) Keras. Keras io.
25. Kasson PM, Jha S (2018) Adaptive ensemble simulations of biomolecules. Current opinion in structural biology 52: 87-94.
26. Roux B, Islam SM (2013) Restrained-ensemble molecular dynamics simulations based on distance histograms from double electron-electron resonance spectroscopy. J Phys Chem B 117: 4733-4739.
27. Hays JM, Kieber MK, Li JZ, Han JI, Moremen KW, et al. (2018) Refinement of highly flexible protein structures using simulation-guided spectroscopy. Angew Chem Int Ed Engl 57: 17110-17114.

28. Hays JM, Cafiso DS, Kasson PM (2019) Hybrid Refinement of Heterogeneous Conformational Ensembles Using Spectroscopic Data. J Phys Chem Lett: 3410-3414.
29. Do Kwon Y, Pancera M, Acharya P, Georgiev IS, Crooks ET, et al. (2015) Crystal structure, conformational fixation and entry-related interactions of mature ligand-free HIV-1 Env. Nature Structural & Molecular Biology 22: 522-531.
30. Stadtmueller BM, Bridges MD, Dam K-M, Lerch MT, Huey-Tubman KE, et al. (2018) DEER Spectroscopy Measurements Reveal Multiple Conformations of HIV-1 SOSIP Envelopes that Show Similarities with Envelopes on Native Virions. Immunity 49: 235-246.e234.
31. Hadjidoukas PE, Bartezzaghi A, Scheidegger F, Istrate R, Bekas C, et al. (2020) torcpy: Supporting task parallelism in Python. SoftwareX 12: 100517.