MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations

Richard J. Gowers[‖**†], Max Linke[‡‡†], Jonathan Barnoud[§†], Tyler J. E. Reddy[‡], Manuel N. Melo[§], Sean L. Seyler[¶], Jan Domanski[‡], David L. Dotson[¶], Sébastien Buchoux[††], Ian M. Kenney[¶], Oliver Beckstein[¶*]

Abstract—MDAnalysis (http://mdanalysis.org) is a library for structural and temporal analysis of molecular dynamics (MD) simulation trajectories and individual protein structures. MD simulations of biological molecules have become an important tool to elucidate the relationship between molecular structure and physiological function. Simulations are performed with highly optimized software packages on HPC resources but most codes generate output trajectories in their own formats so that the development of new trajectory analysis algorithms is confined to specific user communities and widespread adoption and further development is delayed. MDAnalysis addresses this problem by abstracting access to the raw simulation data and presenting a uniform object-oriented Python interface to the user. It thus enables users to rapidly write code that is portable and immediately usable in virtually all biomolecular simulation communities. The user interface and modular design work equally well in complex scripted workflows, as foundations for other packages, and for interactive and rapid prototyping work in IPython / Jupyter notebooks, especially together with molecular visualization provided by nglview and time series analysis with pandas. MDAnalysis is written in Python and Cython and uses NumPy arrays for easy interoperability with the wider scientific Python ecosystem. It is widely used and forms the foundation for more specialized biomolecular simulation tools. MDAnalysis is available under the GNU General Public License v2.

Index Terms—molecular dynamics simulations, science, chemistry, physics, biology

Introduction

Molecular dynamics (MD) simulations of biological molecules have become an important tool to elucidate the relationship between molecular structure and physiological function. Simulations are performed with highly optimized software packages on HPC resources but most codes generate output trajectories in their own formats so that the development of new trajectory analysis algorithms is confined to specific user communities and widespread adoption and further development is delayed. Typical trajectory sizes range from gigabytes to terabytes so it is typically not feasible to convert trajectories into a range of different formats just to use a tool that requires this specific format. Instead, a framework is required that provides a common interface to raw simulation data. Here we describe the MDAnalysis library [MADWB11] that addresses this problem by abstracting access to the raw simulation data. MDAnalysis presents a uniform object-oriented Python interface to the user. Since its original publication in 2011 [MADWB11], MDAnalysis has been widely adopted and has undergone substantial changes. Here we provide a short introduction to MDAnalysis and its capabilities and provide and overview over recent improvements.

† *These authors contributed equally.*
‖ *University of Manchester, Manchester, UK*
** *University of Edinburgh, Edinburgh, UK*
‡‡ *Max Planck Institut für Biophysik, Frankfurt, Germany*
§ *University of Groningen, Groningen, The Netherlands*
‡ *University of Oxford, Oxford, UK*
¶ *Arizona State University, Tempe, Arizona, USA*
†† *Université de Picardie Jules Verne, Amiens, France*
* *Corresponding author: oliver.beckstein@asu.edu*

Overview

MDAnalysis is written in Python and Cython and uses NumPy arrays [VCV11] for easy interoperability with the wider scientific Python ecosystem. Although the primary dependency is NumPy, other Python packages such as netcdf4 and BioPython [HM03] also provide specialized functionality to the core of the library (Figure 1).

MDAnalysis currently supports more than 25 different file formats and covers the vast majority of data formats that are used in the biomolecular simulation community, including the formats required and produced by the most popular packages NAMD, Amber, Gromacs, CHARMM, LAMMPS, DL_POLY, HOOMD. The user interface provides "physics-based" abstractions (e.g. "atoms", "bonds", "molecules") of the data that can be easily manipulated by the user. It hides the complexity of accessing data and frees the user from having to implement the details of different trajectory and topology file formats (which by themselves are often only poorly documented and just adhere to certain "community expectations" that can be difficult to understand for outsiders).

Since the original publication [MADWB11], improvements in speed and data structures make it now possible to work with terabyte-sized trajectories containing up to ~10 million particles. MDAnalysis also comes with specialized analysis classes in the `MDAnalysis.analysis` module that are unique to MDAnalysis such as *LeafletFinder*, a graph-based algorithm for the analysis of lipid bilayers [MADWB11], or *Path Similarity Analysis* for the quantitative comparison of macromolecular conformational changes [SKTB15].

MDAnalysis is available in source form under the GNU General Public License v2 from GitHub as MDAnalysis/mdanalysis, and as PyPi and conda packages. The documentation is extensive and includes an introductory tutorial. The develoment community

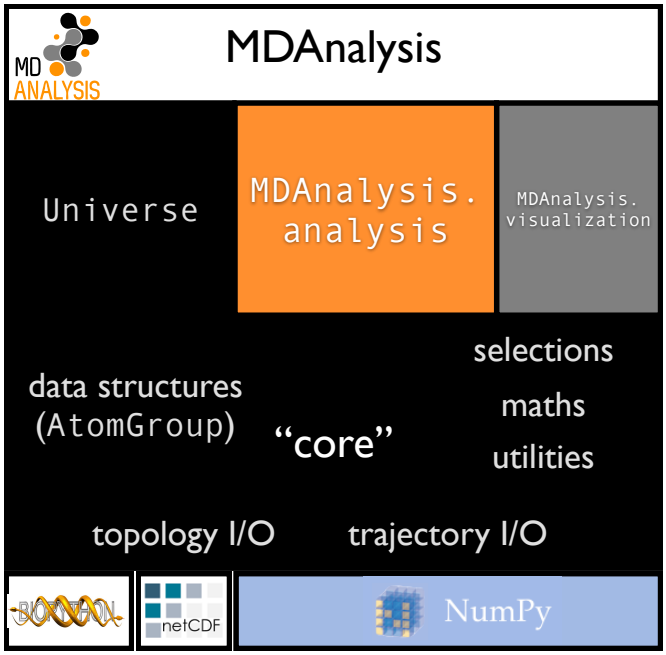

Fig. 1: *Structure of the MDAnalysis package. MDAnalysis consists of the "core" with the* Universe *class as the primary entry point for users. The* MDAnalysis.analysis *package contains independent modules that make use of the core to implement a wide range of algorithms to analyze MD simulations. The* MDAnalysis.visualization *package contains a growing number of tools that are specifically geared towards calculating visual representations such as, for instance, streamlines of molecules.*

is very active with more than five active core developers and many community contributions in every release. We use modern software development practices [WAB+14], [SM14] with continous integration (provided by *Travis CI*) and an extensive automated testsuite (containing over 3500 tests with >92% coverage for our core modules). Development occurs on *GitHub* through pull requests that are reviewed by core developers and other contributors, supported by the results from the automated tests, test coverage reports provided by *Coveralls*, and *QuantifiedCode* code quality reports. Users and developers communicate extensively on the community mailing list (*Google* groups) and the GitHub issue tracker; new users and developers are very welcome. The development and release process is transparent to users. Releases are numbered according to the semantic versioning convention so that users can immediately judge the impact of a new release on their existing code base, even without having to consult the `CHANGELOG` documentation. Old code is slowly deprecated so that users have ample opportunity to update the code although we generally attempt to break as little code as possible. When backwards-incompatible changes are inevitable, we provide tools (based on the Python standard library's *lib2to3*) to automatically refactor code or warn users of possible problems with their existing code.

Basic Usage

The core object in MDAnalysis is the Universe which acts as a nexus for accessing all data contained within a simulation. It is initialised by passing the filenames of the topology and trajectory files, with a multitude of different formats supported in these roles. The topology acts as a description of all the particles in the system while the trajectory describes their behavior over time.

```
import MDAnalysis as mda

# Create a Universe based on simulation results
u = mda.Universe('topol.tpr', 'traj.trr')

# Create a selection of atoms to work with
ag = u.atoms.select_atoms('backbone')
```

The select_atoms method allows for AtomGroups to be created using a human readable syntax which allows queries according to properties, logical statements and geometric criteria.

```
# Select all solvent within a set distance from protein atoms
ag = u.select_atoms('resname SOL and around 5.0 protein')

# Select all heavy atoms in the first 20 residues
ag = u.select_atoms('resid 1:20 and not prop mass < 10.0')

# Use a preexisting AtomGroup as part of another selection
sel1 = u.select_atoms('name N and not resname MET')
sel2 = u.select_atoms('around 2.5 group Nsel', Nsel=sel1)

# Perform a selection on another AtomGroup
sel1 = u.select_atoms('around 5.0 protein')
sel2 = sel1.select_atoms('type O')
```

The AtomGroup acts as a representation of a group of particles, with the properties of these particles made available as NumPy arrays.

```
ag.names
ag.charges
ag.positions
ag.velocities
ag.forces
```

The data from MD simulations comes in the form of a trajectory which is a frame by frame description of the motion of particles in the simulation. Today trajectory data can often reach sizes of hundreds of GB. Reading all these data into memory is slow and impractical. To allow the analysis of such large simulations on an average workstation (or even laptop) MDAnalysis will only load a single frame of a trajectory into memory at any time.

The trajectory data can be accessed through the trajectory attribute of a Universe. Changing the frame of the trajectory object updates the underlying arrays that AtomGroups point to. In this way the positions attribute of an AtomGroup within the iteration over a trajectory will give access to the positions at each frame. Through this approach only a single frame of data is present in memory at any time, allowing for large datasets, from half a million particles [IME+14] to tens of millions, to be dissected with minimal resources.

```
# the trajectory is an iterable object
len(u.trajectory)

# seek to a given frame
u.trajectory[72]
# iterate through every 10th frame
for ts in u.trajectory[::10]:
    ag.positions
```

Example: Per-residue RMSF

As a complete example consider the calculation of the C_α root mean square fluctuation (RMSF) ρ_i that characterizes the mobility of a residue math:i in a protein:

$$\rho_i = \sqrt{\langle (\mathbf{x}_i(t) - \langle \mathbf{x}_i \rangle) \rangle} \qquad (1)$$

A

```
import numpy as np
import MDAnalysis as mda

u = mda.Universe("topol.tpr", "trj.xtc")
ca = u.select_atoms("name CA")
means = np.zeros((len(ca), 3))
sumsq = np.zeros_like(means)
for k, ts in enumerate(u.trajectory):
    sumsq += (k/(k+1.0)) *
             (ca.positions - means)**2
    means[:] = (k*means + ca.positions)/(k+1.0)
rmsf = np.sqrt(sumsq.sum(axis=1)/(k+1.0))

matplotlib.pyplot.plot(ca.residues.resids, rmsf)
```

B

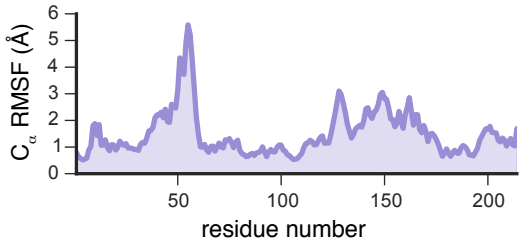

Fig. 2: *Example for how to calculate the root mean square fluctuation (RMSF) for each residue in a protein with MDAnalysis and NumPy. **A**: Based on the input simulation data (topology and trajectory in the Gromacs format (TPR and XTC), MDAnalysis makes coordinates of the selected C_α atoms available as NumPy arrays. From these coordinates, the RMSF is calculated by averaging over all frames in the trajectory. The RMSF is then plotted with `matplotlib`. The algorithm to calculate the variance in a single pass is due to Welford [Wel62]. **B**: C_α RMSF for each residue.*

The code in Figure 2 A shows how MDAnalysis in combination with NumPy can be used to implement Eq. 1. The topology information and the trajectory are loaded into a `Universe` instance; C_α atoms are selected with the MDAnalysis selection syntax and stored as the `AtomGroup` instance `ca`. The main loop iterates through the trajectory using the MDAnalysis trajectory iterator. The coordinates of all selected atoms become available in a NumPy array `ca.positions` that updates for each new time step in the trajectory. Fast operations on this array are then used to calculate variance over the whole trajectory. The final result is plotted with matplotlib as the RMSF over the residue numbers, which are conveniently provided as an attribute of the `AtomGroup` (Figure 2 B).

The example demonstrates how the abstractions that MDAnalysis provides enable users to write very concise code where the computations on data are cleanly separated from the task of extracting the data from the simulation trajectories. These characteristics make it easy to rapidly prototype new algorithms. In our experience, most new analysis algorithms are developed by first prototyping a simple script (like the one in Figure 2), often inside a Jupyter notebook (see section Interactive Use and Visualization). Then the code is cleaned up, tested and packaged into a module. In section Analysis Module, we describe the analysis code that is included as modules with MDAnalysis.

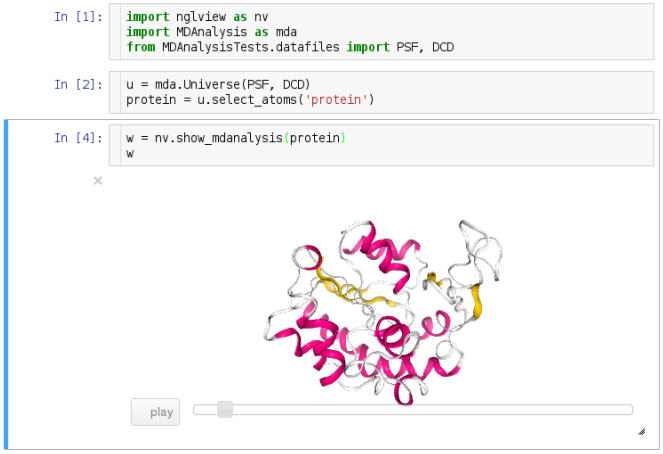

Fig. 3: *MDAnalysis can be used with nglview to directly visualize molecules and trajectories in Jupyter notebooks. The adenylate kinase (AdK) protein from one of the included test trajectories is shown. .*

Interactive Use and Visualization

The high level of abstraction and the pythonic API, together with comprehensive Python doc strings, make MDAnalysis well suited for interactive and rapid prototyping work in IPython [PG07] and Jupyter notebooks. It works equally well as an interactive analysis tool, especially with Jupyter notebooks, which then contain an executable and well-documented analysis protocol that can be easily shared and even accessed remotely. Universes and AtomGroups can be visualized in Jupyter notebooks using nglview, which interacts natively with the MDAnalysis API (Figure 3).

Other Python packages that have become extremely useful in notebook-based analysis workflows are pandas [McK10] for rapid analysis of time series analysis, distributed for simple parallelization, FireWorks [JOC+15] for complex workflows, and MDSynthesis for organizing, bundling and querying many simulations.

Analysis Module

In the `MDAnalysis.analysis` module we provide a large variety of standard analysis algorithms, like RMSD (root mean square distance) and RMSF (root mean square fluctuation) calculations, RMSD-optimized structural superposition [LAT10], native contacts [BHE13], [FKDD07], or analysis of hydrogen bonds as well as unique algorithms, such as the *Leaftleft-Finder* in `MDAnalysis.analysis.leaflet` [MADWB11] and *Path Similarity Analysis* (`MDAnalysis.analysis.psa`) [SKTB15]. Historically these algorithms were contributed by various researchers as individual modules to satisfy their own needs but this lead to some fragmentation in the user interface. We have recently started to unify the interface to the different algorithms with an *AnalysisBase* class. Currently `PersistenceLength`, `InterRDF`, `LinearDensity` and `Contacts` analysis have been ported. `PersistenceLength` calculates the persistence length of a polymer, `InterRDF` calculates the pairwise radial distribution function inside of a molecule, `LinearDensity` generates a density along a given exis and `Contacts` analysis native contacts, as described in more detail below. The API to these different algorithms is being unified with a common `AnalysisBase` class, with an emphasis on keeping it as generic and universal as possible so that it becomes easy to, for instance, parallelize analysis. Most other tools hand the user analysis

algorithms as black boxes. We want to avoid that and allow the user to adapt an analysis to their needs.

The new `Contacts` class is a good example a generic API that allows easy adaptations of algorithms while still offering an easy setup for standard analysis types. The `Contacts` class is calculating a contact map for atoms in a frame and compares it with a reference map using different metrics. The used metric then decides which quantity is measued. A common quantity is the fraction of native contacts, where native contacts are all atompairs that are close to each other in a reference structure. The fraction of native contacts is often used in protein folding to determine when a protein is folded. For native contacts two major types of metrics are considered: ones based on differentiable functions [BHE13] and ones based on hard cut-offs [FKDD07] (which we set as the default implementation). We have designed the API to choose between the two metrics and pass user defined functions to develop new metrics or measure other quantities. This generic interface allowed us to implement a "q1q2" analysis [FKDD07] on top of the `Contacts` class. Below is incomplete code example that shows how to implement a q1q2 analysis, the default value for the *method* kwarg is overwritten with a user defined method *radius_cut_q*. A more detailed explantion can be found in the docs.

```
def radius_cut_q(r, r0, radius):
    y = r <= radius
    return y.sum() / r.size

contacts = Contacts(u, selection,
                    (first_frame, last_frame),
                    radius=radius,
                    method=radius_cut_q,
                    start=start, stop=stop,
                    step=step,
                    kwargs={'radius': radius})
```

This type of flexible analysis algorithm paired with a collection of base classes allow quick and easy analysis of simulations as well as development of new ones.

New data Structures

Originally MDAnalysis followed a strict object-oriented approach with a separate instance of an Atom object for each particle in the simulation data. The AtomGroup then simply stored its contents as a list of these Atom instances. With simulation data commonly containing 10^6 particles this solution did not scale well and so recently this design was overhauled to improve the scalability of MDAnalysis.

Because all Atoms have the same property fields (i.e. mass, position) it is possible to store this information as a single NumPy array for each property. Now an AtomGroup can keep track of its contents as a simple integer array, which can be used to slice these property arrays to yield the relevant data.

Overall this approach means that the same number of Python objects are created for each Universe, with the number of particles only changing the size of the arrays. This translates into a much smaller memory footprint (1.3 GB vs. 3.6 GB for a 10.1 M atom system) highlighting the memory cost of millions of simple Python objects.

This transformation of the data structures from an Array of Structs to a Struct of Arrays also better suits the typical access patterns within MDAnalysis. It is quite common to compare a single property across many Atoms, but rarely are different properties within a single Atom compared. Additionally, it is possible to utilise NumPy's faster indexing rather than using a list comprehension. This new data structure has lead to performance improvements in our whole codebase. The largest improvement is in accessing subsets of Atoms which is now over 40 times faster, see Tables 1, 2 and 3.

# atoms	v0.15.0	v0.16.0	speed up
1.75 M	19	0.45	42
3.50 M	18	0.54	33
10.1 M	17	0.45	38

TABLE 1: *Performance comparison of subselecting an AtomGroup from an existing one using the new system (upcoming release v0.16.0) against the old (v0.15.0). Subselections were slices of the same size (82,056 atoms). Times are given in milliseconds, with shorter times being better. The benchmarks systems were taken from the vesicle library. [KB15] and are listed with their approximate number of particles ("# atoms").*

# atoms	v0.15.0	v0.16.0	speed up
1.75 M	250	35	7.1
3.50 M	490	72	6.8
10.1 M	1500	300	5.0

TABLE 2: *Performance comparison of accessing attributes with new AtomGroup data structures (upcoming release v0.16.0) compared with the old Atom classes (v0.15.0). Times are given in milliseconds, with shorter times being better. The same benchmark systems as in Table 1 were used.*

Other packages that use MDAnalysis

TODO

The user interface and modular design work equally well in complex scripted workflows, as foundations for other packages like ENCORE [TPB+15] and ProtoMD [SMO16].

- MDSynthesis (zenodo REFERENCE), datreant reference

Conclusions

MDAnalysis provides a uniform interface to simulation data, which comes in a bewildering array of formats. It enables users to rapidly write code that is portable and immediately usable in virtually all biomolecular simulation communities. It has a very active international developer community with researchers that are expert developers and users of a wide range of simulation codes. MDAnalysis is widely used (the original paper [MADWB11] has been cited more than 195 times) and forms the foundation

# atoms	v0.15.0	v0.16.0	speed up
1.75 M	18	5	3.6
3.50 M	36	11	3.3
10.1 M	105	31	3.4

TABLE 3: *Performance comparison of loading a topology file with 1.75 to 10 million atoms with new AtomGroup data structures (upcoming release v0.16.0) compared with the old Atom classes (v0.15.0). Loading times are given in seconds, with shorter times being better. The same benchmark systems as in Table 1 were used.*

for more specialized biomolecular simulation tools. Ongoing and future developments will improve performance further, introduce transparent parallelisation schemes to utilize multi-core systems efficiently, and interface with the SPIDAL library for high performance data analytics algorithms.

Acknowledgements

RG was supported by BBSRC grant BB/J014478/1. ML was supported by the Max Planck Society. TR was supported by the Canadian Institutes of Health Research, the Wellcome Trust, the Leverhulme Trust, and Somerville College. Computational resources were provided by PRACE, HPC-Europa2, CINES (France), and the SBCB unit (Oxford). SLS was supported in part by a Wally Stoelzel Fellowship from the Department of Physics at Arizona State University. DLD was in part supported by a Molecular Imaging Fellowship from the Department of Physics at Arizona State University IMK was supported by a REU supplement to grant ACI-1443054 from the National Science Foundation. OB was supported in part by grant ACI-1443054 from the National Science Foundation. JD was in part supported by a Wellcome Trust grant 092970/Z/10/Z.

REFERENCES

[BHE13] Robert B Best, Gerhard Hummer, and William A Eaton. Native contacts determine protein folding mechanisms in atomistic simulations. *Proc. Natl. Acad. Sci. U. S. A.*, 110(44):17874–9, 2013. URL: http://www.pnas.org/content/110/44/17874, doi:10.1073/pnas.1311599110.

[FKDD07] Joel Franklin, Patrice Koehl, Sebastian Doniach, and Marc Delarue. MinActionPath: Maximum likelihood trajectory for large-scale structural transitions in a coarse-grained locally harmonic energy landscape. *Nucleic Acids Res.*, 35(SUPPL.2):477–482, 2007. doi:10.1093/nar/gkm342.

[HM03] Thomas Hamelryck and Bernard Manderick. PDB file parser and structure class implemented in python. *Bioinformatics*, 19(17):2308–2310, 2003. doi:10.1093/bioinformatics/btg299.

[IME+14] Helgi I Ingólfsson, Manuel N Melo, Floris J Van Eerden, Clement Arnarez, Cesar A López, Tsjerk A Wassenaar, Xavier Periole, Alex H De Vries, D Peter Tieleman, and Siewert J Marrink. Lipid Organization of the Plasma Membrane Lipid Organization of the Plasma Membrane. *J. Am. Chem. Soc.*, 136(41):14554–14559, 2014. URL: http://pubs.acs.org/doi/abs/10.1021/ja507832e, arXiv:nn504795v, doi:10.1021/ja507832e.

[JOC+15] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, Daniel Gunter, and Kristin A. Persson. Fireworks: a dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015. CPE-14-0307.R2. URL: http://dx.doi.org/10.1002/cpe.3505, doi:10.1002/cpe.3505.

[KB15] Ian M. Kenney and Oliver Beckstein. SPIDAL Summer REU 2015: Biomolecular benchmark systems. Technical report, Arizona State University, Tempe, AZ, October 2015. doi:10.6084/m9.figshare.1588804.v1.

[LAT10] Pu Liu, Dimitris K Agrafiotis, and Douglas L. Theobald. Fast Determination of the Optimal Rotational Matrix for Macromolecular Superpositions. *J. Comput. Chem.*, 31(7):1561–1563, 2010. arXiv:NIHMS150003, doi:10.1002/jcc.

[MADWB11] Naveen Michaud-Agrawal, Elizabeth Jane Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *J Comp Chem*, 32:2319–2327, 2011. doi:10.1002/jcc.21787.

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python. *Proc. 9th Python Sci. Conf.*, 1697900(Scipy):51–56, 2010. URL: http://conference.scipy.org/proceedings/scipy2010/mckinney.html.

[PG07] Fernando Pérez and Brian E. Granger. IPython: A system for interactive scientific computing. *Comput. Sci. Eng.*, 9(3):21–29, 2007. doi:10.1109/MCSE.2007.53.

[SKTB15] Sean L. Seyler, Avishek Kumar, M. F. Thorpe, and Oliver Beckstein. Path similarity analysis: A method for quantifying macromolecular pathways. *PLoS Comput Biol*, 11(10):e1004568, 10 2015. URL: http://dx.doi.org/10.1371%2Fjournal.pcbi.1004568, doi:10.1371/journal.pcbi.1004568.

[SM14] Victoria Stodden and Sheila Miguez. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. *Journal of Open Research Software*, 2(1):e21, July 2014. URL: http://openresearchsoftware.metajnl.com/article/view/jors.ay, doi:10.5334/jors.ay.

[SMO16] Endre Somogyi, Andrew Abi Mansour, and Peter J. Ortoleva. ProtoMD: A prototyping toolkit for multiscale molecular dynamics. *Computer Physics Communications*, 202:337 – 350, 2016. URL: http://www.sciencedirect.com/science/article/pii/S0010465516300030, doi:10.1016/j.cpc.2016.01.014.

[TPB+15] Matteo Tiberti, Elena Papaleo, Tone Bengtsen, Wouter Boomsma, and Kresten Lindorff-Larsen. ENCORE: Software for quantitative ensemble comparison. *PLoS Comput Biol*, 11(10):e1004415, 10 2015. URL: http://dx.doi.org/10.1371%2Fjournal.pcbi.1004415, doi:10.1371/journal.pcbi.1004415.

[VCV11] Stefan Van Der Walt, S. Chris Colbert, and Gael Varoquaux. The NumPy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30, 2011. arXiv:1102.1523, doi:10.1109/MCSE.2011.37.

[WAB+14] Greg Wilson, D A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven H D Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, Ben Waugh, Ethan P White, and Paul Wilson. Best practices for scientific computing. *PLoS Biol*, 12(1):e1001745, Jan 2014. doi:10.1371/journal.pbio.1001745.

[Wel62] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962. doi:10.1080/00401706.1962.10490022.