The Journal of Open Source Software

BioSimSpace: An interoperable Python framework for biomolecular simulation

Lester O. Hedges[1], **Antonia S.J.S. Mey**[2], **Charles A. Laughton**[3], **Francesco L. Gervasio**[4], **Adrian J. Mulholland**[5], **Christopher J. Woods**[1], **and Julien Michel**[2]

1 Advanced Computing Research Centre, University of Bristol, UK **2** EaStCHEM School of Chemistry, University of Edinburgh, UK **3** School of Pharmacy, University of Nottingham, UK **4** Department of Chemistry and Institute of Structural and Molecular Biology, University College London, UK **5** Centre for Computational Chemistry, School of Chemistry, University of Bristol, UK

DOI: 10.21105/joss.01831

Software

- Review
- Repository
- Archive

Editor: William Rowe

Reviewers:

- @amritagos
- @richardjgowers

Submitted: 21 October 2019
Published: 22 November 2019

Summary

Biomolecular simulation is a diverse and growing area of research, making important contributions to structural biology and pharmaceutical research (Huggins et al., 2019). Within the community there are a several significant and widely used software packages that have evolved from within various research groups over the past 20 or more years. For example, the molecular dynamics packages AMBER (Case et al., 2005), GROMACS (Abraham et al., 2015), and NAMD (Phillips et al., 2005), which are all capable of running biomolecular simulations for systems consisting of hundreds of thousands of atoms and can be run on hardware ranging from laptops, to graphics processing units (GPUs), to the latest high-performance computing clusters. Since different software packages were developed independently, interoperability between them is poor. In large part this is the result of major differences in the supported file formats, which makes it difficult to translate the inputs and outputs of one program to another. As a consequence, expertise in one package doesn't immediately apply to another, making it hard to share methodology and knowledge between different research communities, as evidenced, for instance, by a recent study on reproducibility of relative hydration free energies across simulation packages (Loeffler et al., 2018). The issue is compounded by the increasing use of biomolecular simulations as components of larger scientific workflows for bioengineering or computer-aided drug design purposes. A lack of interoperability leads to brittle workflows, poor reproducibility, and lock in to specific software that hinders dissemination of biomolecular simulation methodologies to other communities.

Several existing software packages attempt to address this problem: InterMol (Shirts et al., 2016) and ParmEd (Swails, Jason, 2010) can be used to read and write a wide variety of common molecular file formats; ACPYPE (Sousa da Silva & Vranken, 2012) can generate small molecule topologies and parameters for a variety of molecular dynamics engines; MDTraj (McGibbon et al., 2015) and MDAnalysis (Gowers et al., 2016) support reading, writing, and analysis of different molecular trajectory formats; the Atomic Simulation Engine (ASE) handles a wide variety of atomistic simulation tasks and provides interfaces to a range of external packages; and the Cuby (Řezáč, 2016) framework allows access to a range of computational chemistry functionality from external packages, which can be combined into complex workflows through structured input files. Despite their utility, the above packages either have a restricted domain of application, e.g. trajectory files, or require different configuration options or scripts to interface with different external packages. It is not possible to write a *single* script that is *independent* of the underlying software packages installed on the host system.

Within the Collaborative Computational Project for Biomolecular Simulation (CCPBioSim), we have attempted to solve this problem via the introduction of an interoperable framework,

BioSimSpace, that collects together the core functionality of many packages and exposes it through a simple Python API. By not choosing to reinvent the wheel, we can take advantage of all the exsting software within the community, and can easily plug into new software packages as they appear. Our software can convert between many common molecular file formats and automatically find packages available within the environment on which it is run. This allows the user to write portable workflow components that can be run with different input, on different environments, and in completely different ways, e.g. from the command-line, or within a Jupyter notebook running on a cloud server. BioSimSpace builds on ideas explored previously by CCPBioSim during the development of the alchemical free energy calculations software FESetup (Loeffler, Michel, & Woods, 2015) that provides consistent setup of input files for several simulation engines.

Molecular dynamics

One of the core features of BioSimSpace is the ability to set up and run molecular dynamics (MD) simulations. There are a large number of packages that can run MD for biomolecules and BioSimSpace supports several of these: AMBER, GROMACS, and NAMD. BioSimSpace also comes with a bundled MD engine, SOMD, which interfaces with the OpenMM (Eastman, 2017) toolkit to provide GPU acceleration. This means that there is always a fall back in case no other MD engines are installed.

While, broadly speaking, the different MD engines offer a similar range of features, their interfaces are quite different. At the heart of this problem is the incompatibility between the molecular file formats used by the different packages. While they all contain the same information, i.e. how atoms are laid out in space and how they interact with each other, the structure of the files is very different. In order to provide interoperability betwen packages we need to be able to read and write many different file formats, and be able to interconvert between them too.

Features

Parsers

At its core, BioSimSpace is built around a powerful set of file parsers which allow reading and writing of a wide range of molecular file formats. File input/output is provided via the `BioSimSpace.IO` package using parsers from the Sire (Woods, Christopher J., 2013) molecular simulation framework, on top of which BioSimSpace is built. Unlike many other programs, we take the approach that it is the *contents* of the file that defines it format, not the *extension*. As such, we attempt to parse a file with all of our parsers in parallel. Any parser for which the contents of the file is incompatible will be rejected early, with the eventual format of the file determined by the parser that completed without error.

Typically, the information needed to construct a molecular system is split across multiple files, e.g. a *coordinate* file containing the atomic coordinates, and a *topology* file that describes how the atoms within each molecule are bonded together, along with parameters for the potential of the molecular model. To handle this, each of our parsers are assigned as being able to *lead*, or *follow*, or both. Lead parsers are able to initialise a molecular system (typically by constructing the topology), whereas those that follow can add additional information to an existing molecular system. Lead parsers may also be able to follow, such that when multiple lead parsers are associated with a set of files then the one that ultimately leads will be determined by which lead parser is unable to follow. This approach allows us to easily parse molecular information from multiple files, even if those formats aren't typically associated with

each other. As long as the molecular topology corresponding to the information in the files is consistent, then they can be read. For instance, one can initialise a system by reading an AMBER format topology, and obtain the coordinates of the system from a Protein Data Bank (PDB) file.

As files are parsed, records in those files are assigned to a set of *properties* that are associated with molecules in the system, e.g. `charge`, `coordinates`, `element`, etc. While some of these properties are unique to particular parsers, others are shared across formats and are converted to a consistent set of internal units on read. Those properties which represent mathematical expressions are stored using Sire's built in computer algebra system. On write, each parser expects molecules in the system to contain a specific set of properties, which are then extracted and converted in order to generate the appropriate records for the format in question. In this way, a bond record from an AMBER format file can be read into an internal bond expression, which could then be converted to the appropriate GROMACS bond record on write. Figure 1 shows a schematic of the file parsing process.

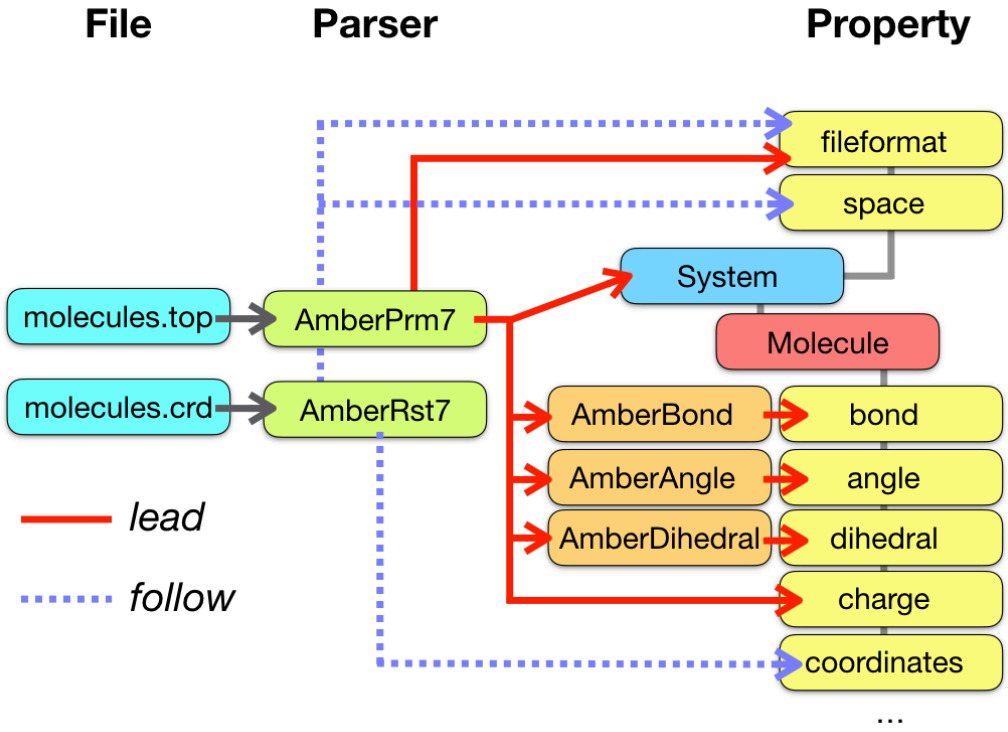

Figure 1: Files are parsed in parallel with the parser that successfully reads the file determining the file format. Once all files are parsed, a lead parser (solid red arrows) constructs the topology of the molecular system. Records within the file, e.g. representing terms in the molecular potential such as bonds, angles, etc., are converted into file format specific representations, then stored internally as properties of the molecule as general algebraic expressions. Parsers that follow add additional information to an existing system. Here the `AmberRst7` parser adds coordinate and simulation box data to the system (dashed blue arrows). The file format associated with the files is also stored as a property of the system so that it is always possible to convert back to the original format on write.

Another feature of our parsers is guaranteed read/write self-consistency. Any file that can be read can also be written, and vice-versa. In addition, when expected molecular information is missing from a file we don't attempt to guess what it may have been. In this sense our parsers don't attempt to be *too* clever, which can lead to unexpected behaviour, particularly when information is modified or supplemented behind the user's back.

The code below shows how to load a set of AMBER format files from a directory:

```
import BioSimSpace as BSS

system = BSS.IO.readMolecules(BSS.IO.glob("amber/ala/*"))
```

Protocols and Processes

BioSimSpace simplifies the set-up and running of molecular simulations through an abstraction of simulation *protocols* and *processes*. Protocols define what a user wants to do to a molecular system, e.g. performing a *minimisation* to find the local potential energy minimum. Processes are used to apply a protocol to a system using a specific molecular simulation engine.

The `BioSimSpace.Protocol` package provides a set of high-level objects for several common molecular simulation protocols. Each protocol offers as set of configurable options that are handled by all of the molecular simulation engines that we support. `BioSimSpace.Process` provides objects for configuring, running, and interacting with simulation processes for each of the supported engines. When a process is created by passing in a system and protocol, BioSimSpace automatically writes all of the input files required by the specific simulation engine and configures any command-line options required by its executable. Expert users of a particular engine are free to fully override any of the configuration options if desired.

The example below shows how to configure and run a default energy minimisation protocol for the molecular system that was loaded earlier. Here we use AMBER as the MD engine:

```
protocol = BSS.Protocol.Minimisation()
process = BSS.Process.Amber(system, protocol)
process.start()
```

Interoperability

While it is useful to be able to configure and run simulation processes using specific engines, any script written in this way would not be portable since we can't guarantee what software will be available on a different computer. To this end, the `BioSimSpace.MD` package provides functionality for automatically configuring a simulation process using *any* suitable MD engine that is installed on the host system. As long as the user has installed an external package using the default installation procedure for that package, or has made sure that the executable is in their shell's path, BioSimSpace will find it. In the case of finding multiple MD engines, BioSimSpace will make a choice based on the file format of the system (to minimise conversions) and whether the user requests GPU support. As an example, the AMBER specific example in the previous section can be translated to an interoperable alternative as follows:

```
protocol = BSS.Protocol.Minimisation()
process = BSS.MD.run(system, protocol)
# By default MD.run starts the process automatically.
```

The `BSS.MD.run` function searches the system for suitable packages that support the chosen protocol, then chooses the most appropriate one to run the simulation. For example, if AMBER was installed then the process returned by `BSS.MD.run` would be of type `BSS.Process.Amber`, if not then the input files could be converted to a different format allowing the use of a different process such as `BSS.Process.Gromacs`.

Robust and flexible workflow components

The building blocks described above can be used to write interoperable workflow components, or *nodes*. Typically, a node will perform a single, well-defined, unit of work with clear inputs and outputs. The `BioSimSpace.Gateway` package acts as a bridge between BioSimSpace and the outside world, allowing a user to construct a node and define the input and output requirements, along with restrictions on their types and values. As an example, the following code snippet shows how the minimisation example described above can be translated into a node.

```
import BioSimSpace as BSS

# Initialise the Node object and set metadata.
node = BSS.Gateway.Node("Minimise a molecular system and save to file.")
node.addAuthor(name="Lester Hedges",
               email="lester.hedges@bristol.ac.uk",
               affiliation="University of Bristol")
node.setLicense("GPLv3")

# Set the inputs and outputs.
node.addInput("files",
    BSS.Gateway.FileSet(help="A set of molecular input files."))
node.addInput("steps",
    BSS.Gateway.Integer(help="The number of minimisation steps.",
                        minimum=0, maximum=1000000, default=10000))
node.addOutput("minimised",
    BSS.Gateway.FileSet(help="The minimised molecular system."))

# Show the graphical user interface (GUI) to allow the user to set the inputs.
# This will only happen if running from within a Jupyter notebook.
node.showControls()

# Load the molecular system and define the a minimisation protocol using the
# user-define input.
system = BSS.IO.readMolecules(node.getInput("files"))
protocol = BSS.Protocol.Minimisation(steps=node.getInput("steps"))

# Execute a simulation process using any available molecular dynamics engine.
process = BSS.MD.run(system, protocol)

# Set the node output to the final configuration of the minimisation process.
# Note that the block=True to the getSystem call to ensure that the
# process finishes before getting the final configuration. (It is possible
# to query the running process in real time when running interactively.)
# Note also that the original file format of the system is preserved on write.
node.setOutput("minimised", BSS.IO.saveMolecules("minimised",
    process.getSystem(block=True), system.fileFormat()))

# Finally, validate the node to make sure that outputs are set correctly
# and no errors have been raised. If running interactively, this will
# generate a download link to a zip file containing the node outputs.
node.validate()
```

BioSimSpace nodes are flexible in the way in which they can be used, with the same script working seamlessly from within a Jupyter notebook or on the command-line. Typically, a user would a write a node as a fully documented, interactive Jupyter notebook, then save it as a regular Python script to run from the command-line. (For inclusion here we simply include the Python script representation of the node, which could be re-converted to a notebook using, e.g., p2j.) Any purely interactive elements included in the node, e.g. visualisations and plots, are simply ignored when the script is run in a non-interactive mode. To facilitate this dual-use the `node.addInput` method generates a custom ipywidgets based graphical user interface for interative use in Jupyter, or a custom argparse parser for handling command-line arguments. Figure 2 shows the example node above running within a Jupyter notebook (top) and from the command-line (bottom).

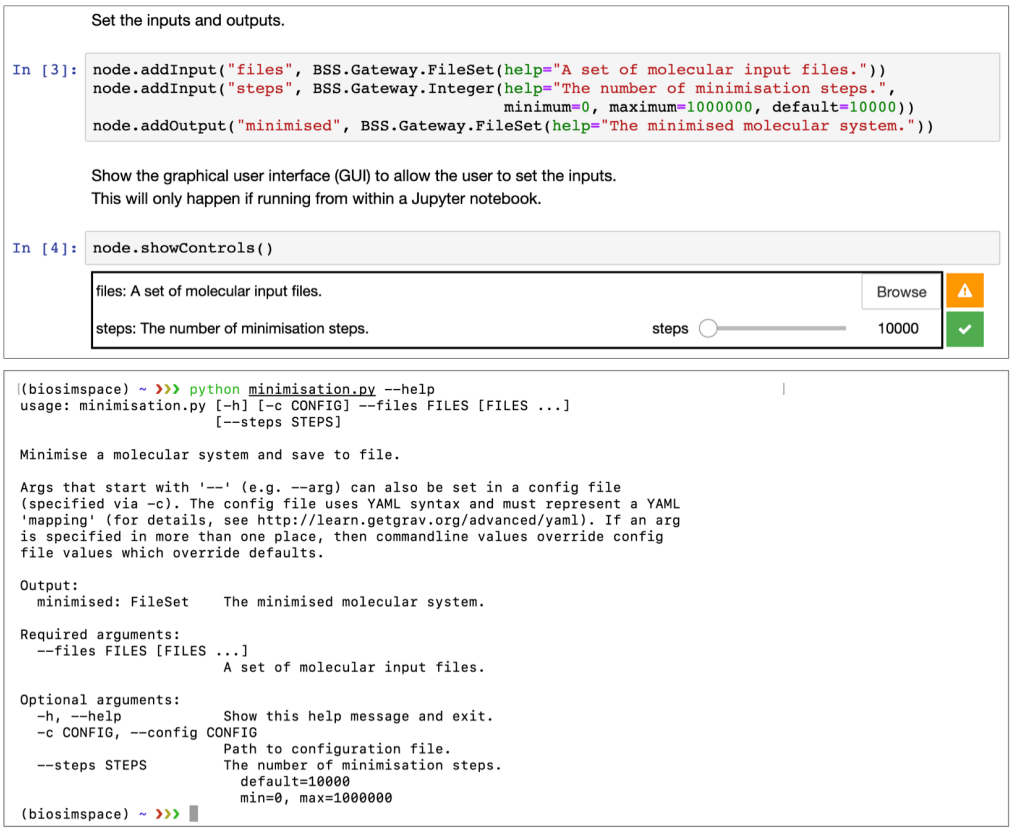

Figure 2: BioSimSpace nodes can be run within a Jupyter notebook (top) or from the command-line (bottom)

When working interactively, BioSimSpace also provides functionality for interacting with processes while they are running. This allows the user to monitor the progress of a simulation and generate near real-time plots and visualisations.

While BioSimSpace isn't intended to be a workflow manager it does provide a means of chaining together nodes by passing the output of one node as the input to another. For example, given the following YAML configuration file, `config.yaml`:

```
files:
- amber/ala.crd
- amber/ala.top
```

it would be possible to run a minimisation followed by an equilibration as follows:

```
python minimisation.py --config config.yaml && \
python equilibration.py --config output.yaml
```

Nodes can also be run from within BioSimSpace itself, allowing the user access to existing functionality as building blocks for more complex scripts. For example, the minimisation node can be run from within BioSimSpace as follows:

```
# Create a dictionary of inputs to the node.
input = {"files" : ["amber/ala.crd", "amber/ala.top"], "steps" : 1000}

# Run the node and capture the output as a dictionary.
output = BSS.Node.run("minimisation", input)
```

Forwards compatibility

To ensure that BioSimSpace nodes are forwards compatible as new features are added all sub packages can query their own functionality and present this to the user. For example, calling `BioSimSpace.IO.fileFormats()` returns a list of the currently supported molecular file formats, `BioSimSpace.Solvent.waterModels()` returns a list of the supported water models, etc. These values can be passed as the `allowed` keyword argument when setting an input requirement of a node, ensuring that the node supports the latest functionality of the package version that is installed. The following code snippet shows a node that can be used to convert to any supported molecular file format, which will continue to work as additional formats are added.

```
import BioSimSpace as BSS

# Initialise the Node object and set metadata.
node = BSS.Gateway.Node("Convert between molecular file formats.")
node.addAuthor(name="Lester Hedges",
               email="lester.hedges@bristol.ac.uk",
               affiliation="University of Bristol")
node.setLicense("GPLv3")

# Set the inputs and outputs and launch the GUI.
node.addInput("files",
    BSS.Gateway.FileSet(help="A set of molecular input files."))
node.addInput("file_format",
    BSS.Gateway.String(help="The format to convert to.",
                       allowed=BSS.IO.fileFormats()))
node.addOutput("converted", BSS.Gateway.File(help="The converted file."))
node.showControls()

# Load the molecular system using the user defined input "files".
system = BSS.IO.readMolecules(node.getInput("files"))

# Convert the system to the chosen format and set the output.
node.setOutput("converted",
    BSS.IO.saveMolecules("converted", system, node.getInput("file_format")))

node.validate()
```

Figure 3 shows how the `allowed=BSS.IO.fileFormats()` argument is translated into a dropdown menu for the Jupyter GUI (top), or using the *choices* option of argparse to display the available options on the command-line (bottom). This means that the script is adaptive to the support of additional file parsers in future without need for modification.

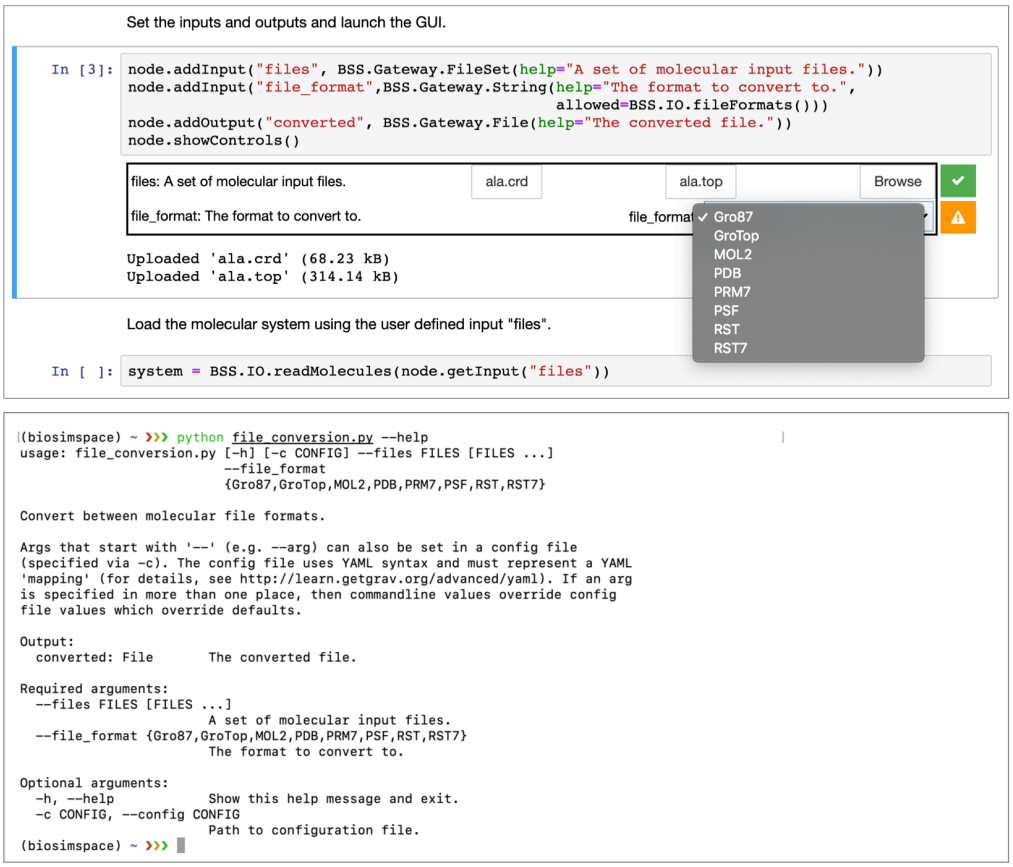

Figure 3: BioSimSpace nodes are adaptive to new functionality without modification.

Extensibility

BioSimSpace has been developed with the intention of being easily extensible. Adding support for a new MD engine just requires the creation of a new `BioSimSpace.Process` class and accompanying functionality to translate the high level `BioSimSpace.Protocol` objects into package specific configuration files. Similarly, it is easy to add support for new tools and utilities as long as they read and write to one of the many molecular file formats that we support. For example, as an alternative to our built in molecular alignment code, it was trivial to wrap the FKCOMBU program from the KCOMBU (Kawabata, 2011) package to enable flexible alignment of molecules. Importantly, support for new packages doesn't change our core API so that new functionality is exposed to users without breaking existing scripts.

Advanced simulation methods

As well as the basic molecular dynamics protocols described so far, BioSimSpace also supports several advanced biomolecular simulation techniques that can be deployed with modular pipelines of setup, run, and analysis nodes executed with best-practices protocols encoded in the library. Our intention is to make it easier to benchmark complex biomolecular simulation

techniques by varying different setup tools, simulation engines, or analysis techniques. For instance, Free-Energy Perturbation (FEP) (Cournia, Allen, & Sherman, 2017) functionality in BioSimSpace can currently be used to compute drug binding affinities with the SOMD or GROMACS simulation engines. By keeping all setup and analysis protocols identical any variability in the results can be ascribed to differences in the simulation engines and protocols only. BioSimSpace also provides support for metadynamics (Barducci, Bonomi, & Parrinello, 2011) simulations using PLUMED (Tribello, Bonomi, Branduardi, Camilloni, & Bussi, 2014) and GROMACS. The application of BioSimSpace FEP and metadynamics workflows to proteins of pharmaceutical interest will be reported elsewhere in due course.

Ease of use

BioSimSpace is avaiable to install from source, as a binary, and as a conda package, all of which are continually built and deployed as part of our developent pipeline. This means that it is easy for users to keep up to date with the latest features, without having to wait for a new release. In addition, access to BioSimSpace is always available through our notebook server, where users are free to work through tutorials and workshop material and make use of our existing repository of nodes.

Acknowledgments

BioSimSpace is the flagship software project of CCPBioSim and was funded through an EPSRC Flagship Software grant: EP/P022138/1. CJW is funded by an EPSRC Research Software Engineering Fellowship: EP/N018591/1.

References

Abraham, M. J., Murtola, T., Schulz, R., Páll, S., Smith, J. C., Hess, B., & Lindahl, E. (2015). GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, *1-2*, 19–25. doi:https://doi.org/10.1016/j.softx.2015.06.001

Barducci, A., Bonomi, M., & Parrinello, M. (2011). Metadynamics. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, *1*(5), 826–843. doi:10.1002/wcms.31

Case, D. A., Cheatham III, T. E., Darden, T., Gohlke, H., Luo, R., Merz Jr., K. M., Onufriev, A., et al. (2005). The amber biomolecular simulation programs. *Journal of Computational Chemistry*, *26*(16), 1668–1688. doi:10.1002/jcc.20290

Cournia, Z., Allen, B., & Sherman, W. (2017). Relative binding free energy calculations in drug discovery: Recent advances and practical considerations. *Journal of Chemical Information and Modeling*, *57*(12), 2911–2937. doi:10.1021/acs.jcim.7b00564

Eastman, J. A. C., Peter AND Swails. (2017). OpenMM 7: Rapid development of high performance algorithms for molecular dynamics. *PLOS Computational Biology*, *13*(7), 1–17. doi:10.1371/journal.pcbi.1005659

Gowers, Linke, Barnoud, Reddy, Melo, Seyler, Domański, et al. (2016). MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations. In Sebastian Benthall & Scott Rostrup (Eds.), *Proceedings of the 15th Python in Science Conference* (pp. 98–105). doi:10.25080/Majora-629e541a-00e

Huggins, D. J., Biggin, P. C., Dämgen, M. A., Essex, J. W., Harris, S. A., Henchman, R. H., Khalid, S., et al. (2019). Biomolecular simulations: From dynamics and mechanisms to

computational assays of biological activity. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, *9*(3), e1393. doi:10.1002/wcms.1393

Kawabata, T. (2011). Build-up algorithm for atomic correspondence between chemical structures. *Journal of Chemical Information and Modeling*, *51*(8), 1775–1787. doi:10.1021/ci2001023

Loeffler, H. H., Bosisio, S., Duarte Ramos Matos, G., Suh, D., Roux, B., Mobley, D. L., & Michel, J. (2018). Reproducibility of free energy calculations across different molecular simulation software packages. *Journal of Chemical Theory and Computation*, *14*(11), 5567–5582. doi:10.1021/acs.jctc.8b00544

Loeffler, H. H., Michel, J., & Woods, C. J. (2015). FESetup: Automating setup for alchemical free energy simulations. *Journal of Chemical Information and Modeling*, *55*(12), 2485–2490. doi:10.1021/acs.jcim.5b00368

McGibbon, R. T., Beauchamp, K. A., Harrigan, M. P., Klein, C., Swails, J. M., Hernández, C. X., Schwantes, C. R., et al. (2015). MDTraj: A modern open library for the analysis of molecular dynamics trajectories. *Biophysical Journal*, *109*(8), 1528–1532. doi:10.1016/j.bpj.2015.08.015

Phillips, J. C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., et al. (2005). Scalable molecular dynamics with namd. *Journal of Computational Chemistry*, *26*(16), 1781–1802. doi:10.1002/jcc.20289

Řezáč, J. (2016). Cuby: An integrative framework for computational chemistry. *Journal of Computational Chemistry*, *37*(13), 1230–1237. doi:10.1002/jcc.24312

Shirts, M. R., Klein, C., Swails, J. M., Yin, J., Gilson, M. K., Mobley, D. L., Case, D. A., et al. (2016). Lessons learned from comparing molecular dynamics engines on the sampl5 dataset. *Journal of computer-aided molecular design*, 1–15. doi:doi:10.1007/s10822-016-9977-1

Sousa da Silva, A. W., & Vranken, W. F. (2012). ACPYPE - antechamber python parser interfacE. *BMC research notes*, *5*, 367–367. doi:10.1186/1756-0500-5-367

Swails, Jason. (2010). ParmEd: Cross-program parameter and topology file editor and molecular mechanical simulator engine. https://github.com/ParmEd/ParmEd.

Tribello, G. A., Bonomi, M., Branduardi, D., Camilloni, C., & Bussi, G. (2014). PLUMED 2: New feathers for an old bird. *Computer Physics Communications*, *185*(2), 604–613. doi:https://doi.org/10.1016/j.cpc.2013.09.018

Woods, Christopher J. (2013). Sire: An advanced, multiscale, molecular simulation framework. http://siremol.org.