Technical Report: SPIDAL Summer REU 2021 Upgrading MDPOW and Adding Analysis Functionality

Alia Lescoulie

Department of Chemistry and Biochemistry, College of Science and Mathematics, California Polytechnic State University, San Luis Obispo, CA, 93407

October 11, 2021

Abstract

MDPOW is a Python package for calculating water-octanol and water-cyclohexane partition coefficients using the molecular dynamics package GROMACS. During the SPIDAL REU, MDPOW was updated to support Python 3. In addition a collection of classes were constructed, simplifying development of simulation analyses. These objects were designed for efficient and organized storage of a group of molecular dynamics systems, and provide a basic framework for developing analyses.

1 Introduction

Molecular dynamics (MD) is one method for computing water-octanol partition coefficients (P_{ow}) of drug-like molecules. This is accomplished through calculating the free energy of solvation from the simulation. With free energy values from water and octanol the partition coefficient can be obtained using $P_{ow} = (\Delta G_w - \Delta G_o)/(RT)\log(e)$ where $R = 8.31446261815 \times 10^{-3}\,\mathrm{kJ/mol \cdot K}$ is the universal gas constant and e is Euler's number [4].

A number of methods exist for obtaining free energies of systems in MD simulations. The two discussed here are Bennett Acceptance Ratio (BAR) and Thermodynamic Integration (TI). The free energy change ΔA from the canonical partition function Q is defined by the following [2]:

$$\Delta A = -\frac{\ln Q_1/Q_0}{\beta} \tag{1}$$

BAR overcomes the challenge of calculating partition functions by substituting the ratio of the partition functions Q_1/Q_0 for the ratio of probabilities (M_0/M_1) for a trial move which maintains the same configurational space but switches potential energies from the reference system U_0 to the unknown system U_1 and vice-versa [1]. The potential energy can be calculated with system positions.

Thermodynamic integration (TI) obtains changes in free energy by defining a parameter λ such that at $\lambda = 0$ $U = U_0$ and at $\lambda = 1$ $U = U_1$, where U_1 is the final state and U_0 is the initial state, and taking the partial derivative of A with respect to λ and integrating from 0 to 1 [2].

Both BAR and TI use λ as a parameter for the extent of solute-solvent interactions [2]. This requires running a MD simulation for each interaction at their respective λ values [6]. The process of initializing that number of simulations requires either the creation of a shell script running each or a tedious process of navigating directories, copying files, and running simulations. MDPOW makes this process easier with the user simply defining their settings and supplying a coordinate file, then running a few Python scripts.

MDPOW is not without its limitations. Before August 2021, it was incompatible with anything newer than Python 2.7 and had few features for analysis beyond running BAR and TI calculations. Both of these features were added over the course of the SPIDAL REU.

Incompatibility with Python 3 created a number of challenges, in particular compatibility with newer versions of dependencies such as MDAnalysis, which dropped Python 2 compatibility [7, 5]. This made new development more difficult, and meant that users had to create an environment for MDPOW separate from the environment used for other portions of a project. For version 0.7.0,

retaining Python 2 cross-compatibility was necessary due to the long-term nature of scientific projects. This meant that many of the new features added in Python 3 such as f-strings, and in Python 3.6 and later, type annotations, were unavailable during the update process [**Beckstein˙Iorga˙2012**, 4].

An additional shortcoming of MDPOW addressed during the REU was its lack of a simple methodology for analyzing simulations. MDPOW, when running free energy perturbation (FEP) simulations, establishes a directory like the example in Fig 1, with simulation files sorted by solvent, interaction and λ-value. Running a simulation to get the partition coefficient can result in forty or more individual systems. For example using MDAnalysis `Universe`s to load systems, one would have to write code to individually load each simulation each in its own directory. That's not to mention actually running analyses on that set of systems. Requiring users to each develop their own analysis tools for managing a larger number of systems would result in lost time and repeated work. Providing an analysis framework makes the process more user friendly.

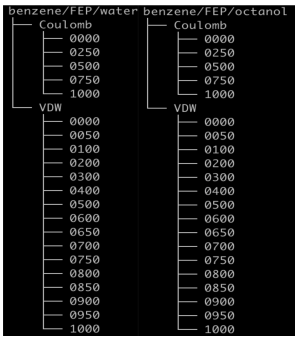

Figure 1: Example MDPOW FEP directory structure

2 Methods

2.1 Modernizing MDPOW

The differences between Python 2 and Python 3 presented compatibility issues in a number of key areas primarily: outdated tests, pickle compatibility issues, and changes to syntax. The first issue was ensuring that all tests ran in both Python versions so that cross compatibility could be validated. MDPOW testing is done in pytest, which eliminated yield tests in version 4.0 requiring their replacement. This was done using parameterization tests where the test parameter and a list of inputs are provided to a function. In the process of running tests, it became apparent reading pickle files was an incompatibility between versions. Pickle files, which save an instance of a Python object as a file, are used throughout MDPOW, particularly in testing. Pickles encoded in Python 2.7, when read normally using the six library in Python 3 raised byte errors. To solve this, pickle loading code was replaced with the block below throughout MDPOW.

```
if sys.version_info.major >= 3:
  with open(gsolv, 'rb') as f:
    G = pickle.load(f, encoding='latin1')
elif sys.version_info.major == 2:
  G = pickle.load(gsolv.open())
```

By checking for the version, pickles can be loaded in byte mode when code is run in Python 3, ensuring compatibility. Patches like this, where separate blocks of code are executed based on version were avoided, but in this case was unavoidable. In MDPOW version 0.8.0 Python 2 support was dropped, and this style of patch has been replaced. The tests and pickle file represented the bulk of compatibility issues. Aside from that the remaining issues were simply the result of syntax differences present in older version of Python.

2.2 Implementation of Analysis Features

Addressing the aforementioned challenge of managing several simulations the `Ensemble` object was developed. The `Ensemble` object is a collection of MDAnalysis `Universe`s stored in a Python dictionary. Python dictionaries, a type of hash table, have O(1) average search efficiency. This means that the time required to find an item is independent of the size of the table [3]. This search efficiency, and the organization created by the key-value pair structure of dictionaries, were the reasons for their selection when developing the `Ensemble` object. An `Ensemble` object is able to load the trajectory files present in a MDPOW molecules directory, automatically handling the process of directory navigation and system loading. The `Ensemble` object also replicates some of the functionality present in MDAnalysis `Universe` objects, namely it allows users to select atoms from the systems it stores. This returns a `EnsembleAtomGroup` object.

The `EnsembleAtomGroup` is to the MDAnalysis AtomGroup as the `Ensemble` is to the MDAnalysis `Universe`. It is initialized by the select atoms command of Ensemble. It, like the `Ensemble` attempts

to extend the functionality of MDAnalysis objects to collections of those objects. It has class methods to select atoms, return positions, and return the original `Ensemble` object.

```
benzene_dir = os.path.join('mdpow', 'tests', 'testing_resources',
                           'states', 'benzene')
Benzene = mdpow.ensemble.Ensemble(dirname=benzene_dir)
DihedralGroup = Benzene.select_atoms('name C1 or name C2 or name C3 or name C4')
```

With the `Ensemble` and `EnsembleAtomGroup` objects an analysis framework analogous to AnalysisBase in MDAnalysis was developed for running calculations on collection of systems generated by MDPOW. Setting up an analysis for a MDPOW simulation requires simply sub classing `EnsembleAnalysis` as in the next listing.

```
class ExampleAnalysis(mdpow.ensemble.EnsembleAnalysis):
  def __init__(self, ExampleEnsemble):
    super(ExampleAnalysis, self).__init__(ExampleEnsemble.ensemble())
    self._ensemble = ExampleEnsemble

  def _prepare_ensemble(self):
    self._cols = ['solvent', 'interaction', 'lambda', 'time', 'result']
    self._result_dict = {x: [] for x in self._cols}
    self.results = pd.DataFrame(column=self._cols

  def _single_frame(self):
    result = analysis_function(self._ensemble[self._key])
    res_list = [self._key[0], self._key[1], self._key[2], self._ts.time, result]
    for i in range(len(res_list)):
      self._result_dict[self._col[i]].append(self.res_list[i])

  def _conclude_ensemble(self):
    for k in self._result_dict:
      self.results[k] = self._result_dict[k]
```

`EnsembleAnalysis` was developed to be similar to AnalysisBase in MDAnalysis. When developing an analysis several aspects must be defined; an `__init__`, which accepts parameters in this case lines 2-4. Among the parameters must be at least one Ensemble, which must be passed back to the parent class using the `super` function from Python as seen in line 3. Next `_prepare_ensemble` and `_prepare_universe` can be used to establish data structures used in the overall analysis and the individual systems. In lines 6-9 on the example a results dictionary and `DataFrame` are set up for organizing results by solvent, interaction, lambda, and time. Next are the methods responsible for generating results from the MD simulations, `_single_universe` which runs on each system in the `Ensemble`, and `_single_frame` which runs on each frame of each `Universe`. `_conclude_universe` and `_conclude_ensemble` are run after each `Universe` and the `Ensemble` respectively. In the about example lines When an `EnsembleAnalysis` based object is run, each system stored in the provided `Ensemble` is iterated over with `_single_universe` ran at that point, and `_single_frame` run on each frame of the `Universe`. In this case `_single_frame` consists of an example `example_function` which returns result on line 12. That result then in the subsequent lines saved into the results dictionary. All of the information from each frame measure is recorded to ensure that the data is tidy. Afterwards there is the option to define `_conclude_universe` which runs each time a new system is completed allowing for processing to occur after each system. Finally after iteration is complete `_conclude_ensemble` is run. In this case it saves the contents of the results dictionary into the results `DataFrame`, this is done at the end because dictionaries are far simpler to add data to within the program, but `DataFrame` is more convenient for saving and plotting data. In the final `DataFrame` each index has a all the information, 'solvent', 'interaction', 'lambda', 'time', and 'result'. This allows for easy organization of Data by multiple parameters for example returning all the results for water and VDW.

3 Results and Discussion

With the establishment of a framework for analyzing collections of simulations, such as those generated by FEP calculations, the process of developing analyses for MDPOW simulations is simplified. These are developed using objects created as subclasses of EnsembleAnalysis.

3.1 Implemented EnsembleAnalysis Methods

Using the `EnsembleAnalsis` framework two new methods were developed as part of the new MDPOW analysis submodule. The first to be discussed is `SolvationAnalysis` which quantifies solvent molecules within the given cutoff distances. It is discussed in section 3.2.1 specifically focusing on how to design an `EnsembleAnalysis` based class. The second is `DihedralAnalysis` in section 3.2.2 which discusses considerations for efficient design of `EnsembleAnalysis` classes.

3.1.1 Solvation Shell: Example for the EnsembleAnalysis Framework

One of the analyses developed with the `EnsembleAnalysis` framework, `SolvationAnalysis`, which returns the number of solvents within the given distances, can also serve as an example of how to develop an analysis more generally. The concept of inherence from object oriented programming is applied to simplify the process.

Building `EnsembleAnalysis` from inheritance, simplifying the process of programming an analysis. When a subclass inherits from a parent class it retains the methods and attributes of that parent. A subclass though doesn't have to remain a carbon copy of the parent. `EnsembleAnalysis` contains several methods which the user can override to run their own code on the systems of the Ensemble. Though not all methods need to be rewritten, rather than redeveloping a method for iterating over a set of systems, and the frames of those systems each time an analysis is developed, a subclass of `EnsembleAnalysis` simply retains those methods from the parent class.

To develop `SolvationAnalysis`, only five new methods, replacing those in the parent class had to be developed. The development of each will be explained below.

```
1  class SolvationAnalysis(EnsembleAnalysis):
2      def __init__(self, solute: EnsembleAtomGroup , solvent: EnsembleAtomGroup ,
       distances: List[float]):
3          self.check_groups_from_common_ensemble([solute, solvent])
4          super(SolvationAnalysis, self).__init__(solute.ensemble)
5          self._solute = solute
6          self._solvent = solvent
7          self._dists = distances
8
9      def _prepare_ensemble(self):
10         self._col = ['distance', 'solvent', 'interaction',
11                      'lambda', 'time', 'N_solvent']
12         self.results = pd.DataFrame(columns=self._col)
13         self._res_dict = {key: [] for key in self._col}
14
15     def _single_frame(self):
16         solute = self._solute[self._key]
17         solvent = self._solvent[self._key]
18         pairs, distances = capped_distance(solute.positions, solvent.positions,
19                                            max(self._dists), box=self._ts.dimensions)
20         solute_i, solvent_j = np.transpose(pairs)
21         for d in self._dists:
22             close_solv_atoms = solvent[solvent_j[distances < d]]
23             result = [d, self._key[0], self._key[1],self._key[2],
24                       self._ts.time, close_solv_atoms.n_atoms]
25             for i in range(len(self._col)):
26                 self._res_dict[self._col[i]].append(result[i])
27
28     def _conclude_ensemble(self):
29         for k in self._col:
30             self.results[k] = self._res_dict[k]
```

The first method `__init__` accepts arguments needed to generate an instance of the object. In this case the user selected solute and solvent, contained in `EnsembleAtomGroups`, and a list of the distances to be measured. On line 2 the `__init__` ensures that the two `EnsembleAtomGroups` originate from the same `Universe` using a built in method of `EnsembleAnalysis` called `check_groups_from_common_ensemble`. This is important as the the groups in `_single_frame` must both exist in the same `Universe`. Next the `Ensemble` from the aforementioned groups is passed into the parent class with the `super` function in line 4. Finally the items are saved as class attributes (saved in the class, and accessible to other methods). The underscore in front of method and attribute names by Python convention indicates that it is protected.

The next method `_prepare_ensemble` establishes data structures needed in the analysis. The final results are available to the user in a `DataFrame`, but for the sake of efficiency the results are stored in

arrays contained in a dictionary with the same keys as the columns in the `DataFrame` as the analysis is run. On line 13 for brevity dictionary interpretation is give each key a blank array. With structures established for storing the results the next methods can focus on obtaining information from the simulation.

With data structures lined up, the actual number of solvent molecules can be counted with `_single_frame`. This is done using the the capped distances function in MDAnalysis on line 18. This computes a distance array for the given group positions and returns distances in that range [7, 5].

Finally the results dictionary is data is inputed into the results DataFrame. This gives data that is neatly organized and easy to generate figures from.

3.1.2 Dihedral Analysis: Building an EnsembleAnalysis Efficiently

Additionally `DihedralAnalysis` was developed using the `EnsembleAnalysis` framework. `DihedralAnalysis` accepts a list of `EnsembleAtomGroups` and returns the dihedral angles over the course of the trajectory. It accomplishes this using the `calc_dihedrals` function from MDAnalysis [7, 5]. This function is written in Cython, which translates python code to C, which runs exponentially faster than code written in python.

4 Conclusion

The modernization of MDPOW and development of the analysis submodule over the course of the SIPDAL REU increased the usability and utility of the library. The update to Python 3 will ensure compatibility with future versions of dependency libraries, and ensure access to future features of those libraries. Additionally the `Ensemble` objects and `EnsembleAnalysis` framework simplify the development of analytical tools within MDPOW.

Acknowledgments

Funding was provided by the National Science Foundation for a REU supplement to award ACI1443054.

References

[1] Charles H Bennett. "Efficient estimation of free energy differences from Monte Carlo data". en. In: *Journal of Computational Physics* 22.2 (Oct. 1976), pp. 245–268. ISSN: 00219991. DOI: 10.1016/0021-9991(76)90078-4. URL: https://linkinghub.elsevier.com/retrieve/pii/0021999176900784 (visited on 06/18/2021).

[2] Christophe Chipot and Andrew Pohorille, eds. *Free energy calculations*. Springer Series in Chemical Physics 86. Berlin: Springer, 2007.

[3] Thomas H. Cormen, ed. *Introduction to algorithms*. en. 3rd ed. OCLC: ocn311310321. Cambridge, Mass: MIT Press, 2009. ISBN: 978-0-262-03384-8 978-0-262-53305-8.

[4] Shujie Fan et al. "Precise force-field-based calculations of octanol-water partition coefficients for the SAMPL7 molecules". en. In: 35 (2021), pp. 853–870.

[5] Richard Gowers et al. "MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations". en. In: Austin, Texas, 2016, pp. 98–105. DOI: 10.25080/Majora-629e541a-00e. URL: https://conference.scipy.org/proceedings/scipy2016/oliver_beckstein.html (visited on 08/09/2021).

[6] Justin Lemkul. "From Proteins to Perturbed Hamiltonians: A Suite of Tutorials for the GROMACS-2018 Molecular Simulation Package [Article v1.0]". en. In: *Living Journal of Computational Molecular Science* 1.1 (2019). ISSN: 25756524. DOI: 10.33011/livecoms.1.1.5068. URL: https://www.livecomsjournal.org/article/5068-from-proteins-to-perturbed-hamiltonians-a-suite-of-tutorials-for-the-gromacs-2018-molecular-simulation-package-article-v1-0 (visited on 08/10/2021).

[7] Naveen Michaud-Agrawal et al. “MDAnalysis: A toolkit for the analysis of molecular dynamics simulations”. en. In: *Journal of Computational Chemistry* 32.10 (July 2011), pp. 2319–2327. ISSN: 01928651. DOI: 10.1002/jcc.21787. URL: https://onlinelibrary.wiley.com/doi/10.1002/jcc.21787 (visited on 08/09/2021).