MDSynthesis: a Python package enabling data-driven molecular dynamics research

D. L. Dotson*, O. Beckstein† / *david.dotson@asu.edu, †oliver.beckstein@asu.edu

Arizona State University — Center for Biological Physics at Arizona State University

Studies using molecular dynamics simulations can routinely generate many terabytes of data, perhaps spread over hundreds of individual run trajectories. Oftentimes these trajectories are not just repeats, but instead sample a wide range of different starting configurations, forcefield parameters, macromolecule conformations, mutations, protonation states, etc., which can make data management difficult. Furthermore, because of their size and cost to calculate, it is often necessary to store intermediate data for collective variables of interest. This adds to the complexity of managing data, and serves as a barrier to answering scientific questions.

To address this problem, our lab has developed MDSynthesis, a Python package that handles the tedious and time-consuming logistics of intermediate data storage and retrieval. MDSynthesis makes working with data from many simulations relatively easy – especially important for interactive data exploration.

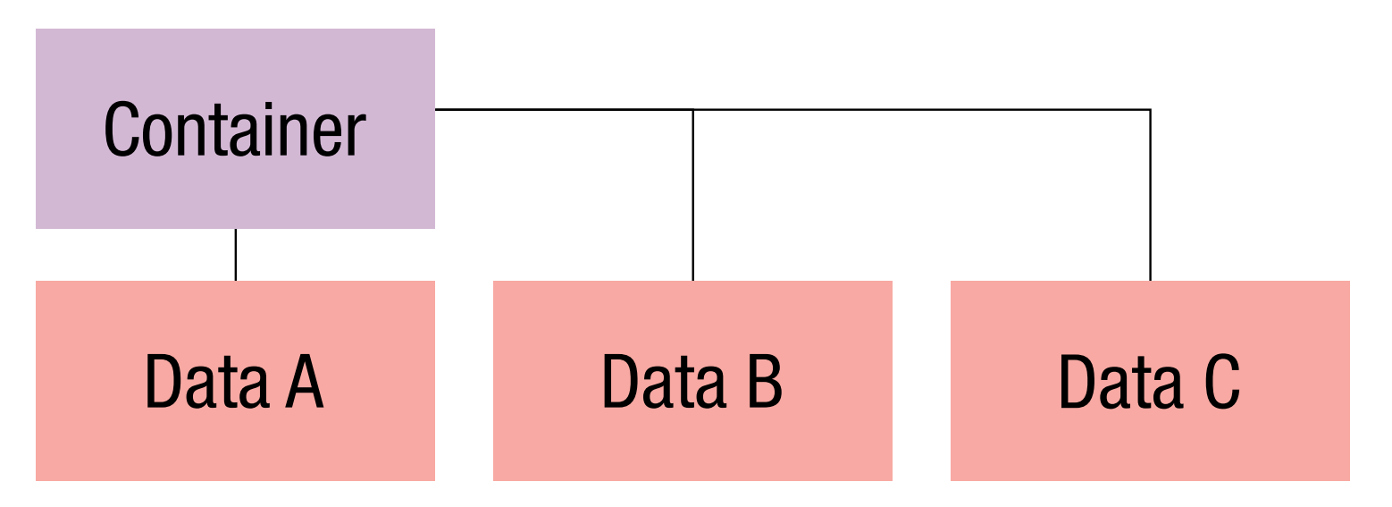
Datasets and Containers

Much of the functionality of MDSynthesis is condensed into two objects, collectively referred to as Containers: the Sim and Group objects.

A Sim is designed to manage and give access to data corresponding to a single simulation, including the raw trajectory(s) and analysis results. A Group gives access to any number of Sims or Groups that it has as members, and it can store analysis results that pertain to these members collectively. Both types of Container store their underlying data persistently to disk on the fly. This allows multiple unrelated processes to make use of these objects simultaneously. For example, making a new Sim named `marklar` is as simple as:

```
>>> import mdsynthesis as mds
>>> s = mds.Sim('marklar')
```

Data storage is the primary function of these Containers. A Container can be thought of as an organizational bin for data that might take a long time to obtain so that it can be easily retrieved later.



```
{ Name: 'marklar'
  Tags(['DIMS', 'gbsw', 'dimer'])
  Categories({'forcefield': 'CHARMM36',
             'transition': 'in to out',
             'protein': 'NhaA'})
```

Figure 1. A Container is a receptacle for data sets. In addition to storing data, Containers can also be differentied with a name, tags, and categories. Containers exist in the filesystem as a directory structure with individual directories for each data set.

Pandas objects (such as Series and DataFrames), Numpy arrays, and other Python data structures can be stored with as little as:

```
>>> s.data['something_wicked'] = data
```

Pandas and Numpy data structures are serialized to disk in the high-performance HDF5 file format, allowing fast out-of-core operations over data sets that may be too large to fit in memory. Objects that can't be serialized into HDF5 are pickled. The data can be retrieved from disk with:

```
>>> data = s.data['something_wicked']
```

Containers and their stored data sets exist as a directory tree on disk. This allows the storage of other files, e.g. figures, with the data. Our example Sim looks like this in the filesystem:

```
marklar/
├── Sim.bc5c5c78-83d5-4164-85b9-e069367ae00a.h5
└── something_wicked
    └── npData.h5
```

Figure 2. The Container API contains the elements common to both Sims and Groups. These include tags and categories, which can be used to differentiate Containers from each other. It also features the data interface.

```
Container
  name -> human-readable identifier
  uuid -> unique identifier
  location -> path to Container in filesystem
  tags -> set of strings describing Container
  categories -> key-value pairs describing Container
  data -> interface to stored data sets
```

Using Sims to dissect trajectories

Sims can do more than store data. They can also store definitions for MDAnalysis[1] Universes, which give an interface to the raw simulation data by way of trajectories on disk. Adding a Universe definition with:

```
>>> s.universes.add('main', topology, trajectory)
```

lets us generate a Universe from the Sim:

```
>>> s.universe
<Universe with 47681 atoms>
```

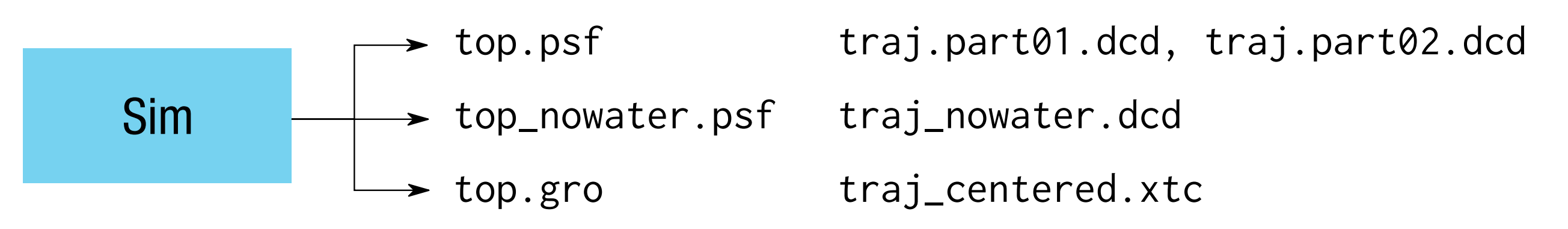


Figure 3. A Sim stores the locations of topologies and trajectories for easy Universe recall. A Sim can store multiple Universe definitions, which are useful when different versions of the same trajectory are needed for different analyses.

Groups of atoms can be selected from a Universe with selection strings. These can be stored by the Sim. We might select all ions in a given simulation with:

```
>>> s.selections['ions'] = 'name NA or name CL'
>>> s.selections['ions']
<<AtomGroup with 178 atoms>
```

This is useful for situations in which many simulations are performed with different forcefields. The selection string needed for a particular group of atoms may differ, but the key used to obtain the stored selection can be the same across them all.

```
Sim (Container)
  universes -> interface to stored universes
  universe -> currently active universe
  selections -> stored selections for active universe
```

Figure 4. The Sim API contains machinery for managing Universe definitions and stored atom selections. A single Sim can store any number of Universe definitions, with different stored selections for each.

Groups are for aggregation

Groups can store data in the same way as Sims, but they also give a convenient interface to working with multiple Sims or Groups at once.

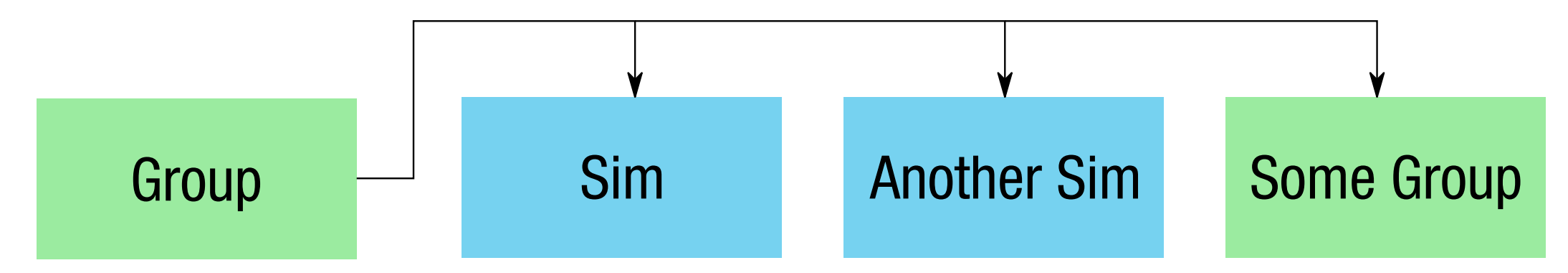


Figure 5. A Group remembers the locations of its members in the filesystem so they can be accessed on demand. A Group has mechanisms to find members that have moved, and will soon be able to retrieve subsets of its members through queries on their attributes, such as tags and categories.

Members of a Group can be accessed directly:

```
>>> g = mds.Group('gruffy', members=[s1, s2, g3])
>>> g
<Group: 'gruffy' | 3 Members: 3 Sims>
>>> g.members[1:]
<Bundle([<Sim: 'fluffy'>, <Group: 'gorp'>])>
```

```
Group (Container)
  members -> interface to group members
  members.tags -> member tags in aggregate
  members.categories -> member categories in aggregate
  members.data -> member data in aggregate
```

Figure 6. The Group API includes an interface for managing members. The Group keeps track of where its members live in the filesystem, and features convenience methods for applying functions in parallel to its members and automatically building aggregates of member data sets.

MD SYNTHESIS built with MD ANALYSIS

Fork me on GitHub

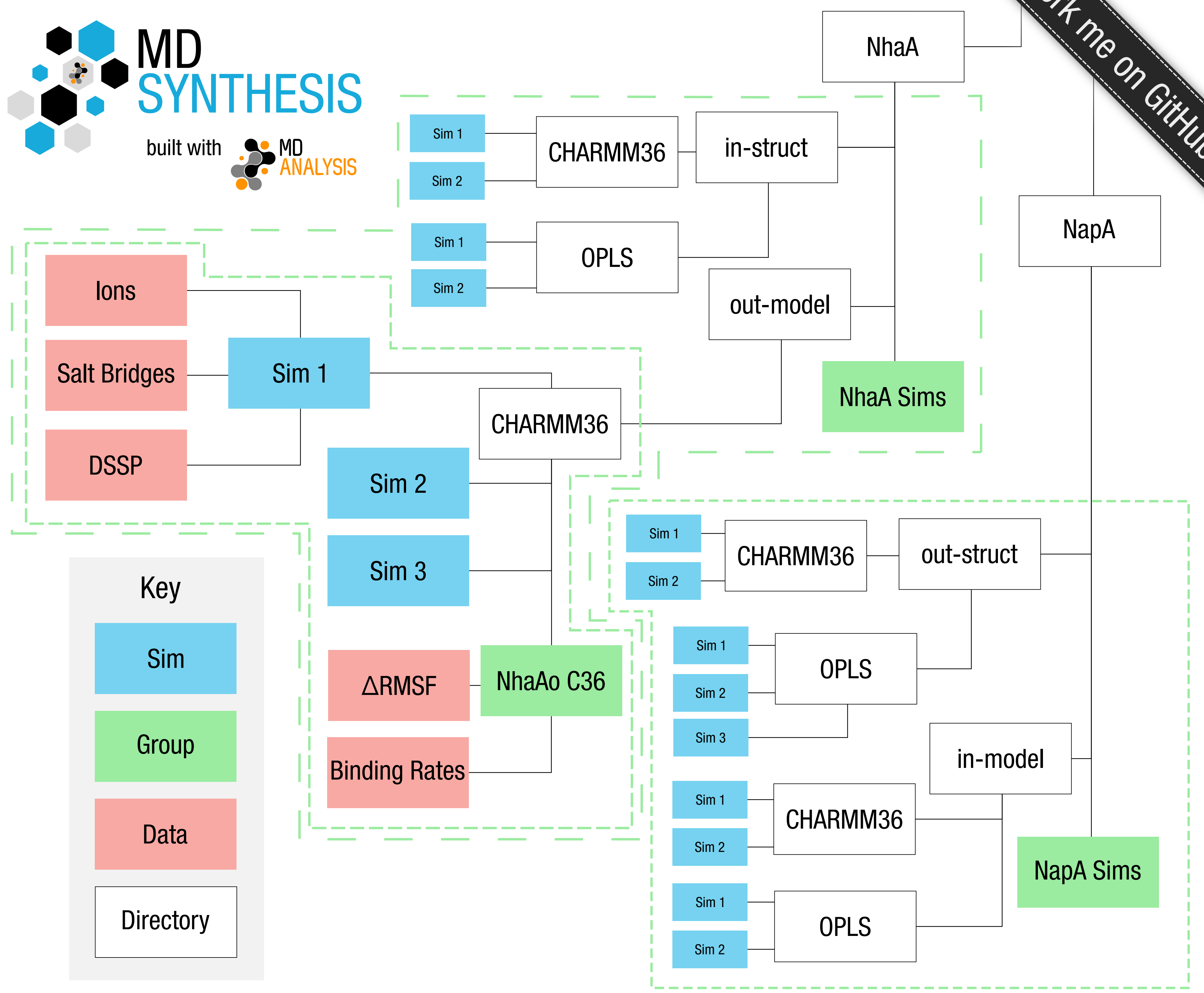


Figure 7. Example of a typical directory hierarchy for simulation work. As more variables are tested, the tree becomes more complex and data management becomes difficult. Sims and Groups may exist anywhere in a directory tree, but they are shown here near where their source data would probably reside. Dashed borders indicate all members included in each Group.

Using Sims and Groups to answer scientific questions quickly

As an example of effective use of Sims and Groups in practice, say we have 50 biased MD simulations sampling the conformational change of the ion transport protein NhaA[2] from inward-open to outward-open. We want to know how many hydrogen bonds exist at any given time between the two domains as they move past each other.

```
import mdsynthesis as mds
from MDAnalysis.analysis.hbonds import HydrogenBondAnalysis
import pandas as pd
import seaborn as sns
g = mds.Group('NhaA_i2o_transitions')

def get_hbonds(sim):
    dimerization = sim.selections.define('dimerization')
    core = sim.selections.define('core')
    hb = HydrogenBondAnalysis(sim.universe, dimerization, core)
    hb.run()
    hb.generate_table()
    sim.data.add('hbonds', pd.DataFrame(hb.table))

g.members.map(get_hbonds, processes=16)
```

Once we've collected the data for each Sim, we can aggregate it, apply a groupby to get the number of bonds present at each time, and then plot the frequency of each number across all simulations in the Group.

```
df = g.members.data.retrieve('hbonds')
counts = df['distance'].groupby(df.index).count()
counts.index = pd.MultiIndex.from_tuples(counts.index)
counts.index = counts.index.droplevel(0)
sns.jointplot(counts.index, counts, kind='hexbin')
```

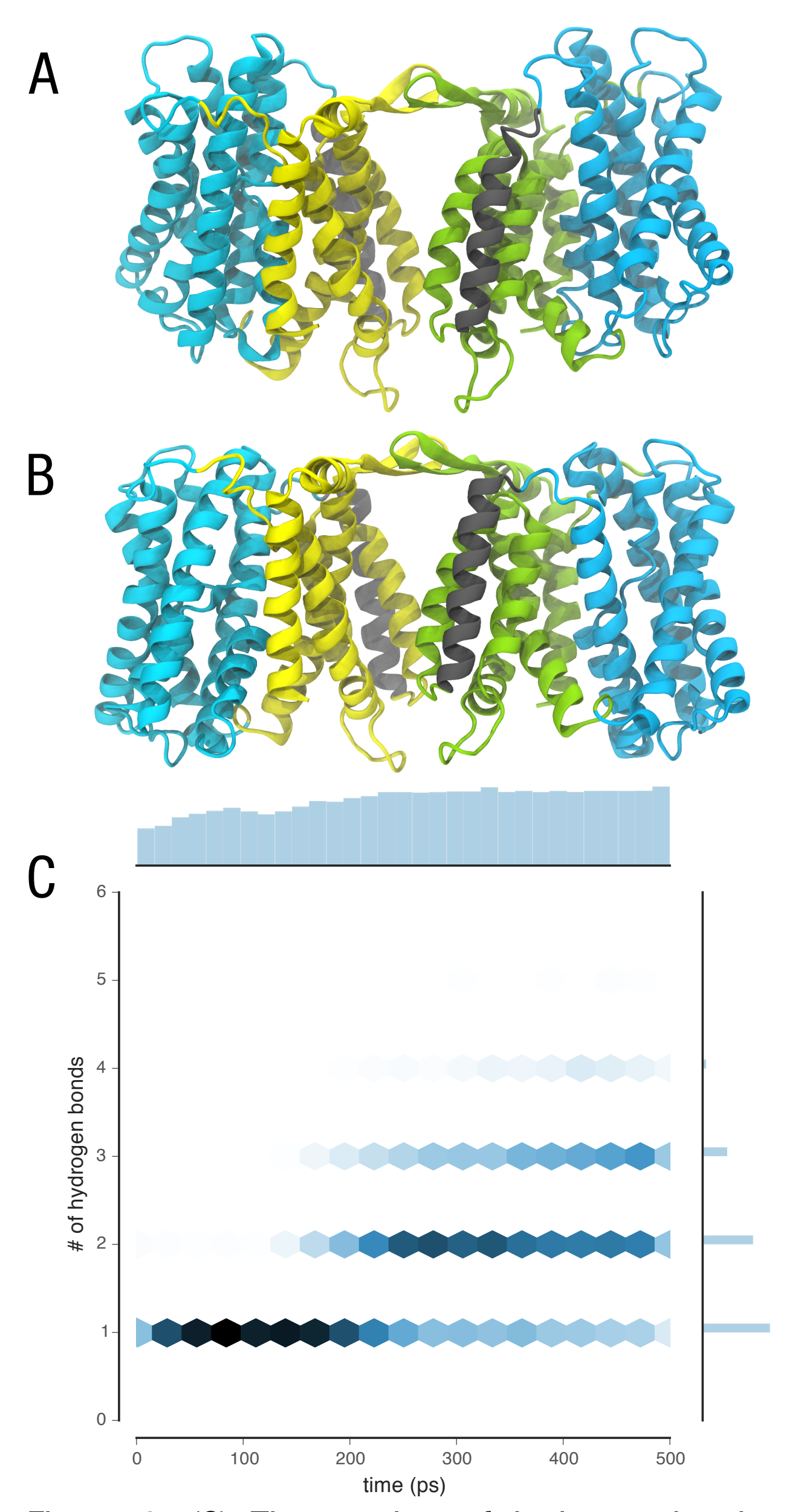


Figure 8. (C) The number of hydrogen bonds between the core (cyan) and dimerization domain (green) during a conformational transition from (B) inward-open to (A) outward-open.

Where to find it

The package is actively developed and freely available under the GNU General Public License at https://github.com/Becksteinlab/MDSynthesis

References

[1] Michaud-Agrawal, N., Denning, E.J., Woolf, T.B., and Beckstein, O. (2011). MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry 32, 2319–2327. http://www.mdanalysis.org

[2] Lee, C., Yashiro, S., Dotson, D.L., Uzdavinys, P., Iwata, S., Sansom, M.S.P., Ballmoos, C. von, Beckstein, O., Drew, D., and Cameron, A.D. (2014). Crystal structure of the sodium-proton antiporter NhaA dimer and new mechanistic insights. J Gen Physiol 144, 529–544.

Acknowledgements

We wish to acknowledge the Arizona State Advanced Computing Center and XSEDE for the use of supercomputing resources.