datreant: persistent, Pythonic trees for heterogeneous data

David L. Dotson[‡][†][*], Sean L. Seyler[‡], Max Linke[§], Richard J. Gowers[¶][‖], Oliver Beckstein[‡]

https://youtu.be/enLHDZoch0U

Abstract—In science the filesystem often serves as a *de facto* database, with directory trees being the zeroth-order scientific data structure. But it can be tedious and error prone to work directly with the filesystem to retrieve and store heterogeneous datasets. **datreant** makes working with directory structures and files Pythonic with **Treants**: specially marked directories with distinguishing characteristics that can be discovered, queried, and filtered. Treants can be manipulated individually and in aggregate, with mechanisms for granular access to the directories and files in their trees. Disparate datasets stored in any format (CSV, HDF5, NetCDF, Feather, etc.) scattered throughout a filesystem can thus be manipulated as meta-datasets of Treants. **datreant** is modular and extensible by design to allow specialized applications to be built on top of it, with MDSynthesis as an example for working with molecular dynamics simulation data. http://datreant.org/

Index Terms—data management, science, filesystems

Introduction

In many scientific fields, especially those analyzing experimental or simulation data, there is an existing ecosystem of specialized tools and file formats which new tools must work around. Consequently, specialized database systems may be unsuitable for data management and storage. In these cases the filesystem ends up serving as a *de facto* database, with directory trees the zeroth-order data structure for scientific data. This is particularly true for fields centered around simulation: simulation systems can vary widely in size, composition, rules, parameters, and starting conditions. And with ever-increasing computational power, it is often necessary to store intermediate results from large amounts of simulation data so that they may be accessed and explored interactively.

These problems make data management difficult, and ultimately serve as a barrier to answering scientific questions. To address this, we present `datreant`, a Pythonic interface to the filesystem. `datreant` deals primarily in **Treants**: specially marked directories with distinguishing characteristics that can be discovered, queried, and filtered. Treants can be manipulated individually and in aggregate, with mechanisms for granular access to the directories and files in their trees. By way of Treants, `datreant` adds a lightweight abstraction layer to the filesystem, allowing researchers to focus more on *what* is stored and less on *where*. This greatly reduces the tedium of storing, retrieving, and operating on datasets of interest, no matter how they are organized.

† *These authors contributed equally.*
∗ *Corresponding author: dldotson@asu.edu*
‡ *Arizona State University, Tempe, Arizona, USA*
§ *Max Planck Institut für Biophysik, Frankfurt, Germany*
¶ *University of Manchester, Manchester, UK*
‖ *University of Edinburgh, Edinburgh, UK*

Copyright © 2016 David L. Dotson et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Treants as filesystem manipulators

The central object of `datreant` is the `Treant`. A Treant is a directory in the filesystem that has been specially marked with a **state file**. A `Treant` is also a Python object. We can create a `Treant` with:

```
>>> import datreant.core as dtr
>>> t = dtr.Treant('maple')
>>> t
<Treant: 'maple'>
```

This creates a directory `maple/` in the filesystem (if it did not already exist), and places a special state file inside which stores the Treant's state. This file also serves as a flagpost indicating that this is more than just a directory:

```
> ls maple
Treant.1dcbb3b1-c396-4bc6-975d-3ae1e4c2983a.json
```

The name of this file includes the type of Treant to which it corresponds, as well as the `uuid` of the Treant, its unique identifier. The state file contains all the information needed to generate an identical instance of this `Treant`, so that we can start a separate Python session and immediately use the same Treant there:

```
# python session 2
>>> import datreant.core as dtr
>>> t = dtr.Treant('maple')
>>> t
<Treant: 'maple'>
```

Making a modification to the `Treant` in one session is immediately reflected by the same `Treant` in any other session. For example, a `Treant` can store any number of descriptive tags to differentiate it from others. We can add tags in the first Python session:

```
# python session 1
>>> t.tags.add('syrup', 'plant')
>>> t.tags
<Tags(['plant', 'syrup'])>
```

And in the other Python session, the same `Treant` with the same tags is visible:

```
# python session 2
>>> t.tags
<Tags(['plant', 'syrup'])>
```

Internally, advisory locking is done to avoid race conditions, making a `Treant` multiprocessing-safe. A `Treant` can also be moved, either locally within the same filesystem or to a remote filesystem, and it will continue to work as expected.

Introspecting a Treant's Tree

A `Treant` can be used to introspect and manipulate its filesystem tree. We can, for example, work with directory structures rather easily:

```
>>> data = t['a/place/for/data/']
>>> data
<Tree: 'maple/a/place/for/data/'>
```

This `Tree` object points to a path in the Treant's own tree, but it need not necessarily exist. We can check this with:

```
>>> data.exists
False
```

This behavior is by design for `Tree` objects (as well as `Leaf` objects; see below). We want to be able to work freely with paths without creating filesystem objects for each, at least until we are ready.

We can make a `Tree` exist in the filesystem easily enough:

```
>>> data.makedirs()
```

and if we also make another directory, too:

```
>>> t['a/place/for/text/'].makedirs()
<Tree: 'maple/a/place/for/text/'>
```

we now have:

```
>>> t.draw()
maple/
 +-- Treant.1dcbb3b1-c396-4bc6-975d-3ae1e4c2983a.json
 +-- a/
     +-- place/
         +-- for/
             +-- data/
             +-- text/
```

Accessing paths in this way returns `Tree` and `Leaf` objects, which refer to directories and files, respectively. These paths need not point to directories or files that actually exist, but they can be used to create and work with these filesystem elements. It should be noted that creating a `Tree` does *not* create a `Treant`. Treants are considered special enough to warrant having a state file with metadata, and making every directory a Treant would make them less useful.

We can, for example, easily store a Pandas [McK10] DataFrame somewhere in the tree for reference later:

```
>>> import pandas as pd
>>> df = pd.DataFrame(pd.np.random.randn(3, 2),
                      columns=['A', 'B'])
>>> data = t['a/place/for/data/']
>>> data
<Tree: 'maple/a/place/for/data/'>
>>> df.to_csv(data['random_dataframe.csv'].abspath)

# take a look at the contents of `data`
>>> data.draw()
data/
+-- random_dataframe.csv
```

and we can introspect the file directly:

```
>>> csv = data['random_dataframe.csv']
>>> csv
<Leaf: 'maple/a/place/for/data/random_dataframe.csv'>

# this should look like a CSV file
>>> print(csv.read())
,A,B
0,-0.573730932177663,-0.08857033924376226
1,0.03157276797041359,-0.10977921690694506
2,-0.2080757315892524,0.6825003213837373
```

Using `Treant`, `Tree`, and `Leaf` objects, we can work with the filesystem Pythonically without giving much attention to precisely *where* these objects live within that filesystem. This becomes especially powerful when we have many directories/files we want to work with, possibly in many different places.

Aggregation and splitting on Treant metadata

What makes a `Treant` distinct from a `Tree` is its **state file**. This file stores metadata that can be used to filter and split `Treant` objects when treated in aggregate. It also serves as a flagpost, making Treant directories discoverable.

If we have many more Treants, perhaps scattered about the filesystem:

```
>>> for path in ('an/elm/', 'the/oldest/oak',
...              'the/oldest/tallest/sequoia'):
...
...     # make a Treant in filesystem at path
...     dtr.Treant(path)
```

we can gather them up with `datreant.core.discover`:

```
>>> b = dtr.discover('.')
>>> b
<Bundle([<Treant: 'oak'>, <Treant: 'sequoia'>,
         <Treant: 'maple'>, <Treant: 'elm'>])>
```

A `Bundle` is an ordered set of `Treant` objects. This collection gives convenient mechanisms for working with Treants as a single logical unit. For example, it exposes a few basic properties for directly accessing its member data:

```
>>> b.relpaths
['the/oldest/oak/',
 'the/oldest/tallest/sequoia/',
 'maple/',
 'an/elm/']

>>> b.names
['oak', 'sequoia', 'maple', 'elm']
```

A `Bundle` can be constructed in a variety of ways, most commonly using existing `Treant` instances or paths to Treants in the filesystem.

We can use a `Bundle` to subselect Treants in typical ways, including integer indexing and slicing, fancy indexing, boolean indexing, and indexing by name. But in addition to these, we can use metadata features such as **tags** and **categories** to filter and group Treants as desired.

Filtering Treants with tags

Tags are individual strings that describe a Treant. Setting the tags for each of our Treants separately:

```
>>> b['maple'].tags = ['syrup', 'furniture', 'plant']
>>> b['sequoia'].tags = ['huge', 'plant']
>>> b['oak'].tags = ['for building', 'plant', 'building']
>>> b['elm'].tags = ['firewood', 'shady', 'paper',
                     'plant', 'building']
```

we can now work with these tags in aggregate:

```
# will only show tags present in *all* members
>>> b.tags
<AggTags(['plant'])>

# will show tags present among *any* member
>>> b.tags.any
{'building',
 'firewood',
 'for building',
 'furniture',
 'huge',
 'paper',
 'plant',
 'shady',
 'syrup'}
```

and we can filter on them. For example, getting all Treants that are good for construction work:

```
# gives a boolean index for members with this tag
>>> b.tags['building']
[True, False, False, True]

# we can use this to index the Bundle itself
>>> b[b.tags['building']]
<Bundle([<Treant: 'oak'>, <Treant: 'elm'>])>
```

or getting back Treants that are both good for construction *and* used for making furniture by giving tags as a list:

```
# a list of tags serves as an *intersection* query
>>> b[b.tags[['building', 'furniture']]]
<Bundle([])>
```

which in this case none of them are.

Other tag expressions can be constructed using tuples (for *or*/*union* operations) and sets (for a *negated intersection*), and nesting of any of these works as expected:

```
# we can get a *union* by using a tuple
>>> b[b.tags['building', 'furniture']]
<Bundle([<Treant: 'maple'>, <Treant: 'oak'>,
         <Treant: 'elm'>])>

# we can get a *negated intersection* by using a set
>>> b[b.tags[{'building', 'furniture'}]]
<Bundle([<Treant: 'sequoia'>, <Treant: 'maple'>,
         <Treant: 'oak'>, <Treant: 'elm'>])>
```

Using tag expressions, we can filter to Treants of interest from a `Bundle` counting many, perhaps hundreds, of Treants as members. A common workflow is to use `datreant.core.discover` to gather up many Treants from a section of the filesystem, then use tags to extract only those Treants one actually needs.

Splitting Treants on categories

Categories are key-value pairs that provide another mechanism for distinguishing Treants. We can add categories to each Treant:

```
# add categories to individual members
>>> b['oak'].categories = {'age': 'adult',
                           'type': 'deciduous',
                           'bark': 'mossy'}
>>> b['elm'].categories = {'age': 'young',
                           'type': 'deciduous',
                           'bark': 'smooth'}
>>> b['maple'].categories = {'age': 'young',
                             'type': 'deciduous',
                             'bark': 'mossy'}
>>> b['sequoia'].categories = {'age': 'old',
                               'type': 'evergreen',
                               'bark': 'fibrous',
                               'home': 'california'}

# add value 'tree' to category 'plant'
# for all members
>>> b.categories.add({'plant': 'tree'})
```

and we can access categories for individual Treants:

```
>>> seq = b['sequoia'][0]
>>> seq.categories
<Categories({'home': 'california',
             'age': 'old',
             'type': 'evergreen',
             'bark': 'fibrous',
             'plant': 'tree'})>
```

The aggregated categories for all members in a `Bundle` are accessible via `Bundle.categories`, which gives a view of the categories with keys common to *every* member Treant:

```
>>> b.categories
<AggCategories({'age': ['adult', 'young',
                        'young', 'old'],
                'type': ['deciduous', 'deciduous',
                         'deciduous', 'evergreen'],
                'bark': ['mossy', 'smooth',
                         'mossy', 'fibrous'],
                'plant': ['tree', 'tree',
                          'tree', 'tree']})>
```

Each element of the list associated with a given key corresponds to the value for each member, in member order. Using `Bundle.categories` is equivalent to `Bundle.categories.all`; we can also access categories present among *any* member:

```
>>> b.categories.any
{'age': ['adult', 'young', 'young', 'old'],
 'bark': ['mossy', 'smooth', 'mossy', 'fibrous'],
 'home': [None, None, None, 'california'],
 'type': ['deciduous', 'deciduous',
          'deciduous', 'evergreen']}
```

Members that do not have a given key will have `None` as the corresponding value in the list. Accessing values for a list of keys:

```
>>> b.categories[['age', 'home']]
[['adult', 'young', 'young', 'old'],
 [None, None, None, 'california']]
```

or a set of keys:

```
>>> b.categories[{'age', 'home'}]
{'age': ['adult', 'young', 'young', 'old'],
 'home': [None, None, None, 'california']}
```

returns, respectively, a list or dictionary of lists of values, where the list for a given key is in member order. Perhaps the most powerful feature of categories is the `groupby` method, which, given a key, can be used to group specific members in a `Bundle` by their corresonding category values. If we want to group members by their `'bark'`, we can use `groupby` to obtain a dictionary of members for each value of `'bark'`:

```
>>> b.categories.groupby('bark')
{'fibrous': <Bundle([<Treant: 'sequoia'>])>,
 'mossy': <Bundle([<Treant: 'oak'>,
                    <Treant: 'maple'>])>,
 'smooth': <Bundle([<Treant: 'elm'>])>}
```

Say we would like to get members grouped by both their 'bark' and 'home':

```
>>> b.categories.groupby({'bark', 'home'})
{('fibrous', 'california'):
    <Bundle([<Treant: 'sequoia'>])>}
```

We get only a single member for the pair of keys ('fibrous', 'california') since 'sequoia' is the only Treant having the 'home' category. Categories are useful as labels to denote the types of data that a Treant may contain or how the data were obtained. By leveraging the groupby method, one can extract Treants by selected categories without having to explicitly access each member. This feature can be particularly powerful in cases where many Treants have been created and categorized to handle incoming data over an extended period of time; one can quickly gather any data needed without having to think about low-level details.

Treant modularity with attachable Limbs

Treant objects manipulate their tags and categories using Tags and Categories objects, respectively. These are examples of Limb objects: attachable components which serve to extend the capabilities of a Treant. While Tags and Categories are attached by default to all Treant objects, custom Limb subclasses can be defined for additional functionality.

datreant is a namespace package, with the dependency-light core components included in datreant.core. The dependencies of datreant.core include backports of standard library modules such as pathlib and scandir, as well as lightweight modules such as fuzzywuzzy and asciitree.

datreant.core remains lightweight because other packages in the datreant namespace can have any dependencies they require. One such package is datreant.data, which includes a set of convenience Limb objects for storing and retrieving Pandas and NumPy [vdW11] datasets in HDF5 using PyTables and h5py internally.

We can attach a Data limb to a Treant with:

```
>>> import datreant.data
>>> t = dtr.Treant('maple')
>>> t.attach('data')
>>> t.data
<Data([])>
```

and we can immediately start using it to store e.g. a Pandas Series:

```
>>> import numpy as np
>>> sn = pd.Series(np.sin(
...     np.linspace(0, 8*np.pi, num=200)))
>>> t.data['sinusoid'] = sn
```

and we can get it back just as easily:

```
>>> t.data['sinusoid'].head()
0    0.000000
1    0.125960
2    0.249913
3    0.369885
4    0.483966
dtype: float64
```

Looking at the directory structure of "maple", we see that the data was stored in an HDF5 file under a directory corresponding to the name we stored it with:

```
>>> t.draw()
maple/
 +-- sinusoid/
 |   +-- pdData.h5
 +-- Treant.1dcbb3b1-c396-4bc6-975d-3ae1e4c2983a.json
```

What's more, datreant.data also includes a corresponding AggLimb for Bundle objects, allowing for automatic aggregation of datasets by name across all member Treant objects. If we collect and store similar datasets for each member in our Bundle:

```
>>> b = dtr.discover('.')
>>> b
<Bundle([<Treant: 'oak'>, <Treant: 'sequoia'>,
         <Treant: 'maple'>, <Treant: 'elm'>])>

# we want to make each dataset a bit different
>>> b.categories['frequency'] = [1, 2, 3, 4]
>>> for mem in b:
...     freq = mem.categories['frequency']
...     mem.data['sinusoid'] = pd.Series(np.sin(
...         freq * np.linspace(0, 8*np.pi, num=200)))
```

then we can retrieve all of them into a single, multi-index Pandas Series:

```
>>> sines = b.data.retrieve('sinusoid', by='name')
>>> sines.groupby(level=0).head()
sequoia  0    0.000000
         1    0.125960
         2    0.249913
         3    0.369885
         4    0.483966
oak      0    0.000000
         1    0.369885
         2    0.687304
         3    0.907232
         4    0.998474
maple    0    0.000000
         1    0.249913
         2    0.483966
         3    0.687304
         4    0.847024
elm      0    0.000000
         1    0.483966
         2    0.847024
         3    0.998474
         4    0.900479
dtype: float64
```

which we can use for aggregated analysis, or perhaps just pretty plots (Figure 1).

```
>>> for name, group in sines.groupby(level=0):
...     s = group.reset_index(level=0, drop=True)
...     s.plot(legend=True, label=name)
```

The Data limb stores Pandas and NumPy objects in the HDF5 format within a Treant's own tree. It can also store arbitrary (but pickleable) Python objects as pickles, making it a flexible interface for quick data storage and retrieval. However, it ultimately serves as an example for how Treant and Bundle objects can be extended to do complex but convenient things.

Using Treants as the basis for dataset access and manipulation with the PyData stack

Although it is possible to extend datreant objects with limbs to do complex operations on a Treant's tree, it isn't necessary

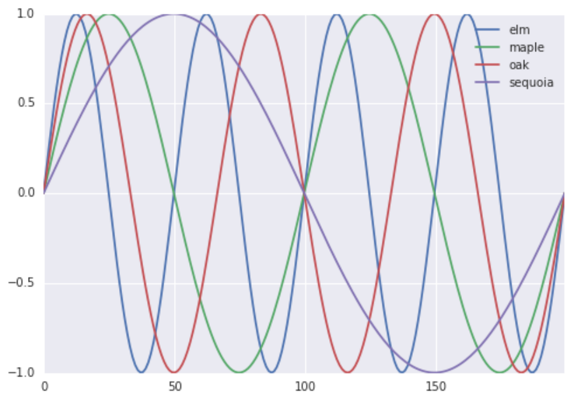

***Fig. 1:** Plot of sinusoidal toy datasets aggregated and plotted by source Treant.*

to build specialized interfaces such as these to make use of the extensive PyData stack. `datreant` fundamentally serves as a Pythonic interface to the filesystem, bringing value to datasets and analysis results by making them easily accessible now and later. As data structures and file formats change, `datreant` objects can always be used in the same way to supplement the way these tools are used.

Because each Treant is both a Python object and a filesystem object, they work remarkably well with distributed computation libraries such as dask.distributed [Roc15] and workflow execution frameworks such as Fireworks [Jai15]. Treant metadata features such as tags and categories can be used for automated workflows, including backups and remote copies to external compute resources, making work on datasets less imperative and more declarative when desired.

Building domain-specific applications on datreant

Built-in `datreant.core` objects are general-purpose, while packages like `datreant.data` provide extensions to these objects that are more specific. But it is possible, and very useful, for domain-specific applications to define their own domain-specific `Treant` subclasses, with tightly-coupled limbs for domain-specific needs. Not only do objects such as `Bundle` work just fine with `Treant` subclasses and custom `Limb` classes; they are designed explicitly with this need in mind.

The first example of a domain-specific package built around `datreant` is MDSynthesis, a module that enables high-level management and exploration of molecular dynamics simulation data. MDSynthesis gives a Pythonic interface to molecular dynamics trajectories using MDAnalysis [MiA11], giving the ability to work with the data from many simulations scattered throughout the filesystem with ease. This package makes it possible to write analysis code that can work across many varieties of simulation, but even more importantly, MDSynthesis allows interactive work with the results from hundreds of simulations at once without much effort.

Leveraging molecular dynamics data with MDSynthesis

MDSynthesis defines a `Treant` subclass called a `Sim`. A `Sim` featues special limbs for storing an MDAnalysis `Universe` definition and custom atom selections within its state file, allowing for painless recall of raw simulation data and groups of atoms of interest.

As an example of effectively using `Sims`, say we have 50 biased molecular dynamics simulations that sample the conformational change of the ion transport protein NhaA [Lee14] from the inward-open to outward-open state (Figure 2). Let's also say that we are interested in how many hydrogen bonds exist at any given time between the two domains as they move past each other. These `Sim` objects already exist in the filesystem, each having a `Universe` definition already set to point to its unique trajectory file(s).

We can use the MDAnalysis `HydrogenBondAnalysis` class to collect the data for each `Sim` using `Bundle.map` for process parallelism, storing the results using the `datreant.data` limb:

```python
import mdsynthesis as mds
import MDAnalysis.analysis.hbonds as hbonds
import pandas as pd
import seaborn as sns

b = mds.discover('NhaA_i2o_transitions')

def get_hbonds(sim):
    dimerization = sim.atomselections['dimer']
    core = sim.atomselections['core']

    hb = hbonds.HydrogenBondAnalysis(
            sim.universe, dimerization, core)
    hb.run()
    hb.generate_table()

    sim.data['hbonds'] = pd.DataFrame(hb.table)

# process parallelism provided internally
# with `multiprocessing`
b.map(get_hbonds, processes=16)
```

Then we can retrieve the datasets in aggregate using the `Bundle` `datreant.data` limb and visualize the result (Figure 3):

```python
df = b.data.retrieve('hbonds', by='name')

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')
```

By making it relatively easy to work with what can often be many terabytes of simulation data spread over tens or hundreds of trajectories, MDSynthesis greatly reduces the time it takes to iterate on new ideas toward answering real biological questions.

Final thoughts

`datreant` is a young project that started as a domain-specific package for working with molecular dynamics data, but has quickly morphed into a powerful, general-purpose tool for managing and manipulating filesystems and the data spread about them. The dependency-light `datreant.core` package is pure Python, BSD-licensed, and openly developed, and the `datreant` namespace is designed to support useful extensions to the core objects. It is the hope of the authors that `datreant` continues to grow in a way that benefits the wider scientific community, smoothing the common pain point of data glut and filesystem management.

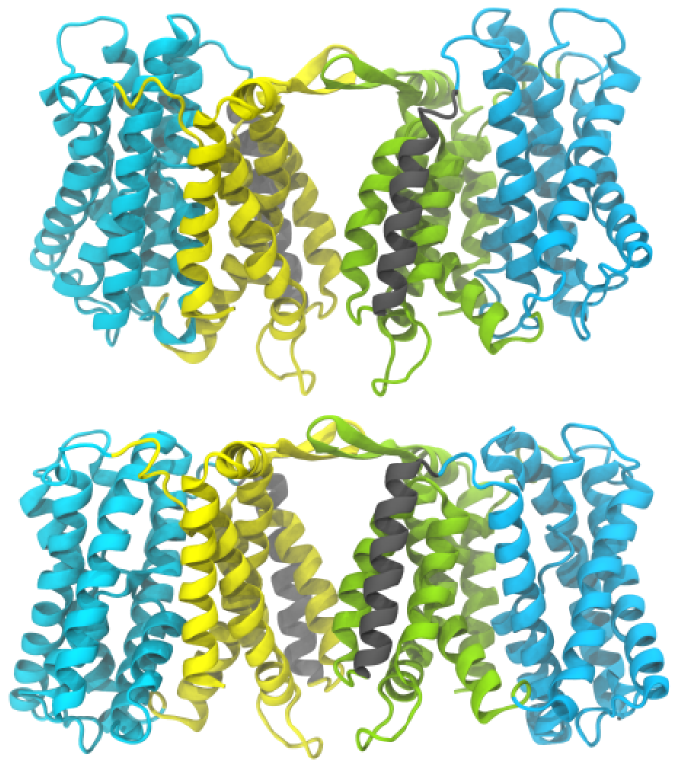

Fig. 2: *A cartoon rendering of an outward-open model (top) and an inward-open crystallographic structure (PDB ID: 4AU5 [Lee14]) (bottom) of* Escherichia coli *NhaA.*

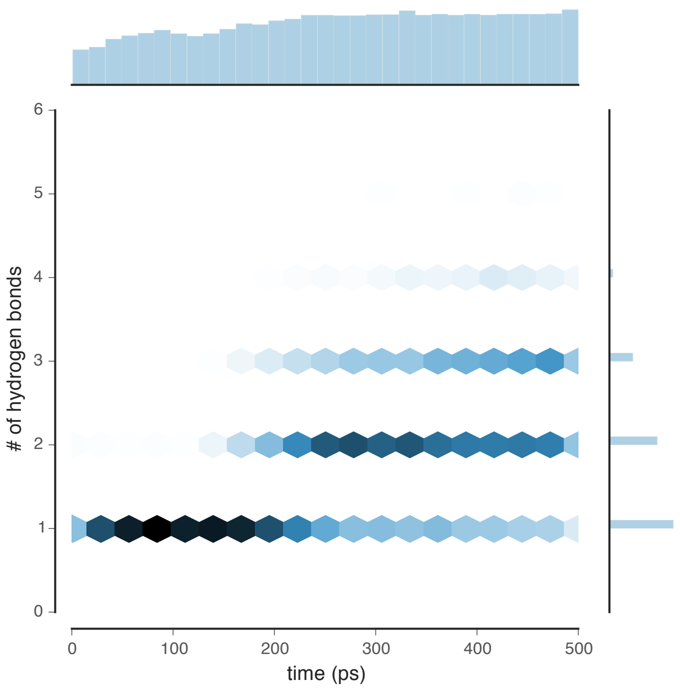

Fig. 3: *The number of hydrogen bonds between the core and dimerization domain during a conformational transition between the inward-open and outward-open state of EcNhaA.*

Acknowledgements

DLD was in part supported by a Molecular Imaging Fellowship from the Department of Physics at Arizona State University. SLS was supported in part by a Wally Stoelzel Fellowship from the Department of Physics at Arizona State University. ML was supported by the Max Planck Society. RG was supported by BBSRC grant BB/J014478/1. OB was supported in part by grant ACI-1443054 from the National Science Foundation; computational resources for OB's work were in part provided by the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575 (allocation MCB130177 to OB).

References

[vdW11] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011)

[Roc15] Matthew Rocklin. Dask: Parallel Computation with Blocked algorithms and Task Scheduling, Proceedings of the 14th Python in Science Conference, 130-136 (2015)

[Jai15] A. Jain, S. P. Ong, W. Chen, B. Medasani, X. Qu, M. Kocher, M. Brafman, G. Petretto, G.-M. Rignanese, G. Hautier, D. Gunter, and K. A. Persson. FireWorks: a dynamic workflow system designed for high-throughput applications. Concurrency Computat.: Pract. Exper., 27: 5037–5059. doi: 10.1002/cpe.3505 (2015)

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

[MiA11] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf and O. Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations, J Comp Chem, 32: 2319-2327. doi: 10.1002/jcc.21787 (2011)

[Lee14] C. Lee, S. Yashiro, D. L. Dotson, P. Uzdavinys, S. Iwata, M. S. P. Sansom, C. von Ballmoos, O. Beckstein, D. Drew, and A. D. Cameron. Crystal structure of the sodium-proton antiporter NhaA dimer and new mechanistic insights, J Gen Physiol, 144:529–544. doi: 10.1085/jgp.201411219 (2014)