MDSynthesis Documentation

Release 0.6.2-dev

David Dotson

April 06, 2017

As computing power increases, it is now possible to produce hundreds of molecular dynamics simulation trajectories that vary widely in length, system size, composition, starting conditions, and other parameters. Managing this complexity in ways that allow use of the data to answer scientific questions has itself become a bottleneck. MDSynthesis is an answer to this problem.

Built on top of datreant, MDSynthesis gives a Pythonic interface to molecular dynamics trajectories using MDAnalysis, giving the ability to work with the data from many simulations scattered throughout the filesystem with ease. It 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.

Warning: This package is **experimental**. It is not API stable, and has many rough edges and limitations. It is, however, usable.

CHAPTER 1

Efficiently store intermediate data from individual simulations for easy recall

The MDSynthesis **Sim** object gives an interface to raw simulation data through MDAnalysis. Data structures generated from raw trajectories (pandas objects, numpy arrays, or any pure python structure) can then be stored and easily recalled later. Under the hood, datasets are stored in the efficient HDF5 format when possible.

CHAPTER 2

Powered by `datreant` under the hood

MDSynthesis is built on top of the general-purpose datreant library. The Sim is a `Treant` with special features for working with molecular dynamics data, but every feature of datreant applies to MDSynthesis.

Getting MDSynthesis

See the installation instructions for installation details. The package itself is pure Python, but many of its dependencies are not.

If you want to work on the code, either for yourself or to contribute back to the project, clone the repository to your local machine with:

```
git clone https://github.com/datreant/MDSynthesis.git
```

Contributing

This project is still under heavy development, and there are certainly rough edges and bugs. Issues and pull requests welcome!

MDSynthesis follows the development model of datreant; see the contributor's guide to learn how to get started with contributing back.

Installing MDSynthesis

Since MDSynthesis requires `datreant.data`, which uses HDF5 as the file format of choice for persistence, you will first need to install the HDF5 libraries either using your package manager or manually.

On **Ubuntu 14.04** this will be

```
apt-get install libhdf5-serial-1.8.4 libhdf5-serial-dev
```

and on **Arch Linux**

```
pacman -S hdf5
```

You can then install MDSynthesis from PyPI using pip:

```
pip install mdsynthesis
```

It is also possible to use `--user` to install into your user's site-packages directory:

```
pip install --user mdsynthesis
```

Be aware that some dependencies may require `numpy` and/or Cython to be installed beforehand!

Installing using Conda

We also provide conda packages. Using the conda packages has the advantage that they contain the hdf5 library, so you can install MDSynthesis easily on all unix systems including macOS. To install MDSynthesis with conda type:

```
conda install -c datreant mdsynthesis
```

Dependencies

The dependencies of MDSynthesis are:

- MDAnalysis: 0.14 or higher
- datreant.core: 0.6.0 or higher
- datreant.data: 0.6.0 or higher

Installing from source

To install from source, clone the repository and switch to the master branch

```
git clone git@github.com:datreant/MDSynthesis.git
cd MDSynthesis
git checkout master
```

Installation of the packages is as simple as

```
pip install .
```

which installs `mdsynthesis` in the system wide python directory (this may require administrative privileges).

It is also possible to use `--user` to install into your user's site-packages directory:

```
pip install --user .
```

Leveraging molecular dynamics data with Sims

A *`Sim`* is a `Treant` with specialized components for working with molecular dynamics data. In particular, it can store a definition for an MDAnalysis `Universe` for painless recall, as well as custom atom selections.

Note: Since Sims are Treants, everything that applies to Treants applies to Sims as well. See the datreant documentation for how Treants work and effectively used.

As with a normal Treant, to generate a Sim from scratch, we need only give it a name

```
>>> from mdsynthesis import Sim
>>> s = Sim('adk')
>>> s
<Sim: 'adk'>
```

And we can immediately give the Sim characteristics like tags:

```
>>> s.tags.add('biased', 'closed-to-open', 'mep')
>>> s.tags
<Tags(['biased', 'closed-to-open', 'MEP'])>
```

and categories:

```
>>> s.categories['sampling method'] = 'DIMS'
>>> s.categories['sampling '] = 'heavy atom'
<Categories({'sampling ': 'heavy atom', 'sampling method': 'DIMS'})>
```

These can be used later to filter and aggregate Sims when we have many to work with. They can also be used as switches for analysis code, since we may need to do different things depending on, for example, the type of sampling method used to produce the trajectory.

Defining the Universe

What makes a Sim different from a basic Treant is that it can store an MDAnalysis `Universe` definition. We can access the Sim's Universe directly with:

```
>>> s.universe
```

At this point, we get back `None`. However, we can define the Sim's Universe by giving it one direclty:

```
>>> import MDAnalysis as mda
>>> s.universe = mda.Universe('path/to/topology.psf',
                              'path/to/trajectory.dcd')
>>> s.universe
<Universe with 3341 atoms and 3365 bonds>
```

The Universe definition is persistent, so we can get back an identical Universe later from another Python session with our Sim:

```
>>> import mdsynthesis as mds
>>> s = mds.Sim('adk')
>>> s.universe
<Universe with 3341 atoms and 3365 bonds>
```

Changing the Universe definition

Warning: This interface may be removed in a future release, but remains for now due to limitations in MDAnalysis. It is encouraged to set the Universe definition directly as shown above.

We can directly change the topology used for the Sim's Universe with

```
>>> s.universedef.topology = 'path/to/another/topology.psf'
```

then we get back a Universe built with this topology instead:

```
>>> s.universe.filename
'/home/bob/research/path/to/another/topology.psf'
```

We can also change the trajectory:

```
>>> s.universedef.trajectory = 'path/to/another/trajectory.dcd'
```

which re-initializes the Universe with both the defined topology and trajectory:

```
>>> s.universe.trajectory
<DCDReader /home/bob/research/path/to/another/trajectory.dcd with 98 frames of 3341 atoms>
```

We can also define our Universe as having multiple trajectories by giving a list of filepaths instead. Internally, the Universe generated will use the MDAnalysis `ChainReader` for treating the trajectories as a contiguous whole.

Note: Changing the topology or trajectory definition will reload the Universe automatically. This means that any AtomGroups you are working with will not point to the new Universe, but perhaps the old one, so it's generally best to regenerate them manually.

Storing keyword arguments

If the Universe needed requires keyword arguments on initialization, these can be stored as well. For example, if our topology was a PDB file and we wanted bonds to be guessed upfront, we could make this happen every time:

```
>>> s.universedef.kwargs = {'guess_bonds': True}
```

Reinitializing the Universe

If you make modifications to the Universe but you want to restore the original from its definition, you can force it to reload with:

```
>>> s.universedef.reload()
```

API Reference: UniverseDefinition

See the *UniverseDefinition* API reference for more details.

Storing custom atom selections

MDAnalysis includes its own selection language for extracting `AtomGroup` objects, which function as ordered lists of (selected) atoms from the system. The selection strings needed to specify these can be long and complex, and sometimes multiple selection strings are required in a particular order to extract a given AtomGroup from all the atoms in the Universe. Moreover, given different simulation systems, the same selection of atoms (e.g. the "solvent") might require a different set of selection strings.

Fortunately, Sims provide a mechanism for storing (many) atom selections. Say we want to select the LID, CORE, and NMP domains of adenylate kinase, the protein we simulated. We can store these immediately:

```
>>> s.atomselections['lid'] = 'resid 122:159'
>>> s.atomselections['nmp'] = 'resid 30:59'
>>> s.atomselections['core'] = ('resid 1:29', 'resid 60:121', 'resid 160:214')
```

We can now get new AtomGroups back for each selection at any time with the `create()` method:

```
>>> s.atomselections.create('lid')
<AtomGroup with 598 atoms>

>>> s.atomselections.create('core')
<AtomGroup with 2306 atoms>
```

and we don't have to remember or know how 'lid' or 'core' are defined for this particular system. If we have other simulations of adenylate kinase performed with other molecular dynamics engines or with different forcefields, we can store the atom selection strings required for those systems in the same way, perhaps using the same names 'lid', 'core', etc. This abstraction allows us to work with many variants of a simulation system without having to micromanage.

Note: Storing a list of strings as a selection will apply them in order, producing an AtomGroup concatenated from each one in the same way as providing multiple strings to `select_atoms()` does. This is especially useful when storing selections used for structural alignments.

Want just the selection strings back? We can use `get()`:

```
>>> s.atomselections.get('lid')
'resid 122:159'

# or using getitem syntax
>>> s.atomselections['lid']
'resid 122:159'
```

Atom selections from atom indices

Do you already have an AtomGroup and prefer to define it according to its atom indices instead of as a selection string? That can be done, too:

```
>>> lid = s.universe.select_atoms('resid 122:159')
>>> s.atomselections['lid'] = lid.indices
>>> s.atomselections.create('lid')
<AtomGroup with 598 atoms>
```

Lists/tuples of selection strings or atom indices can be stored in any combination as a selection. These are applied in order to yield the AtomGroup when calling the `create()` method.

API Reference: AtomSelections

See the *AtomSelections* API reference for more details.

API Reference: Sim

See the *Sim* API reference for more details.

Using Sims with `datreant` objects

Since MDSynthesis is built on top of datreant, many datreant components are exposed in the MDSynthesis namespace.

Functions

`discover`([dirpath, depth, treantdepth])	Find all Treants within given directory, recursively.

Classes

`Treant`(treant[, new, categories, tags])	The Treant: a Tree with a state file.
`Tree`(dirpath[, limbs])	A directory.
`Leaf`(filepath)	A file in the filesystem.
`View`(*vegs, **kwargs)	An ordered set of Trees and Leaves.
`Bundle`(*treants, **kwargs)	An ordered set of Treants.
`Group`(treant[, new, categories, tags])	A Treant with a persistent Bundle of other Treants.

Learning more

See the datreant docs for details on how to put these objects to work with your Sims.

API Reference

This is an overview of the API components included directly within `mdsynthesis`.

Individual simulations

The Sim is the core unit of functionality of `mdsynthesis`. They function as markers for simulation data, giving an interface to raw topologies and trajectories by way of MDAnalysis.

The components documented here are those included within `mdsynthesis`. However, the API elements of *:mod:datreant.core* and *:mod:datreant.data* are also available for use with Sims.

Sim

The class *`mdsynthesis.Sim`* is the central object of `mdsynthesis`.

class `mdsynthesis.`**`Sim`**(*sim*, *new=False*, *categories=None*, *tags=None*)

The Sim object is an interface to data for a single simulation.

sim should be a base directory of a new or existing Sim. An existing Sim will be regenerated if a state file is found. If no state file is found, a new Sim will be created.

A Tree object may also be used in the same way as a directory string.

If multiple Treant state files are in the given directory, a `MultipleTreantsError` will be raised; specify the full path to the desired state file to regenerate the desired Treant in this case. It is generally better to avoid having multiple state files in the same directory.

Use the *new* keyword to force generation of a new Sim at the given path.

Parameters

- **`sim`** (*`str or Tree`*) – Base directory of a new or existing Sim; will regenerate a Sim if a state file is found, but will genereate a new one otherwise; may also be a Tree object
- **`new`** (*`bool`*) – Generate a new Sim even if one already exists at the given location

- **categories** (*dict*) – dictionary with user-defined keys and values; used to give Sims distinguishing characteristics
- **tags** (*list*) – list with user-defined values; like categories, but useful for adding many distinguishing descriptors

abspath
Absolute path of `self.path`.

atomselections
Stored atom selections for the universe.

Useful atom selections can be stored for the universe and recalled later.

attach(**limbname*)
Attach limbs by name to this Treant.

categories
Interface to categories.

children
A View of all files and directories in this Tree.

Includes hidden files and directories.

data
Interface to stored data.

discover(*dirpath='.'*, *depth=None*, *treantdepth=None*)
Find all Treants within given directory, recursively.

Parameters

- **dirpath** (*string, Tree*) – Directory within which to search for Treants. May also be an existing Tree.
- **depth** (*int*) – Maximum directory depth to tolerate while traversing in search of Treants. `None` indicates no depth limit.
- **treantdepth** (*int*) – Maximum depth of Treants to tolerate while traversing in search of Treants. `None` indicates no Treant depth limit.

Returns **found** – Bundle of found Treants.

Return type Bundle

draw(*depth=None*, *hidden=False*)
Print an ASCII-fied visual of the tree.

Parameters

- **depth** (*int*) – Maximum directory depth to display. `None` indicates no limit.
- **hidden** (*bool*) – If False, do not show hidden files; hidden directories are still shown if they contain non-hidden files or directories.

exists
Check existence of this path in filesystem.

filepath
Absolute path to the Treant's state file.

glob(*pattern*)
Return a View of all child Leaves and Trees matching given globbing pattern.

Arguments

pattern globbing pattern to match files and directories with

hidden
A View of the hidden files and directories in this Tree.

leaves
A View of the files in this Tree.

Hidden files are not included.

limbs
A set giving the names of this object's attached limbs.

loc
Get Tree/Leaf at *path* relative to Tree.

Use with getitem syntax, e.g. `.loc['some name']`

Allowed inputs are: - A single name - A list or array of names

If directory/file does not exist at the given path, then whether a Tree or Leaf is given is determined by the path semantics, i.e. a trailing separator ("/").

Using e.g. `Tree.loc['some name']` is equivalent to doing `Tree['some name']`. `.loc` is included for parity with `View` and `Bundle` API semantics.

location
The location of the Treant.

Setting the location to a new path physically moves the Treant to the given location. This only works if the new location is an empty or nonexistent directory.

make()
Make the directory if it doesn't exist. Equivalent to `makedirs()`.

Returns This Tree.

Return type Tree

makedirs()
Make all directories along path that do not currently exist.

Returns

tree this tree

name
The name of the Treant.

The name of a Treant need not be unique with respect to other Treants, but is used as part of Treant's displayed representation.

parent
Parent directory for this path.

path
Treant directory as a `pathlib2.Path`.

relpath
Relative path of `self.path` from current working directory.

sync(*other*, *mode='upload'*, *compress=True*, *checksum=True*, *backup=False*, *dry=False*, *include=None*, *exclude=None*, *overwrite=False*, *rsync_path='/usr/bin/rsync'*)
Synchronize directories using rsync.

Parameters

- **other** (*str or Tree*) – Other end of the sync, can be either a path or another Tree.
- **mode** (*str*) – Either `"upload"` if uploading to *other*, or `"download"` if downloading from *other*

The other options are described in the `datreant.core.rsync.rsync()` documentation.

tags
Interface to tags.

treants
Bundle of all Treants found within this Tree.

This does not return a Treant for a bare state file found within this Tree. In effect this gives the same result as `Bundle(self.trees)`.

treanttype
The type of the Treant.

tree
This Treant's directory as a Tree.

trees
A View of the directories in this Tree.

Hidden directories are not included.

universe
The universe of the Sim.

Universes are interfaces to raw simulation data, with stored selections for this universe directly available via `Sim.selections`.

Setting this to a `MDAnalysis.Universe` will set that as the universe definition for this Sim. Setting to `None` will remove the universe definition entirely.

universedef
The universe definition for this Sim.

uuid
Get Treant uuid.

A Treant's uuid is used by other Treants to identify it. The uuid is given in the Treant's state file name for fast filesystem searching. For example, a Treant with state file:

```
'Treant.7dd9305a-d7d9-4a7b-b513-adf5f4205e09.h5'
```

has uuid:

```
'7dd9305a-d7d9-4a7b-b513-adf5f4205e09'
```

Changing this string will alter the Treant's uuid. This is not generally recommended.

Returns

uuid unique identifier string for this Treant

walk (*topdown=True*, *onerror=None*, *followlinks=False*)
Walk through the contents of the tree.

For each directory in the tree (including the root itself), yields a 3-tuple (dirpath, dirnames, filenames).

Parameters

- **topdown** (*Boolean, optional*) – If False, walks directories from the bottom-up.

- **onerror** (*function, optional*) – Optional function to be called on error.
- **followlinks** (*Boolean, optional*) – If False, excludes symbolic file links.

Returns Wrapped *scandir.walk()* generator yielding *datreant* objects

Return type generator

UniverseDefinition The class *mdsynthesis.limbs.UniverseDefinition* is the interface used by a Sim to define its `MDAnalysis.Universe`.

class `mdsynthesis.limbs.`**`UniverseDefinition`**(*treant*)

The defined universe of the Sim.

Universes are interfaces to raw simulation data, with stored selections for this universe directly available via `Sim.atomselections`.

`kwargs`

The keyword arguments applied to the Sim's universe when building it.

Set these with a dictionary of keywords and values to change them. Keywords must be strings and values must be strings, ints, floats, bools, or `None`.

`reload`()

Re-load the universe from its stored definition.

`topology`

The topology file for this Sim's universe.

To change the topology file for this Sim's universe, set the path to the file. Setting to `None` will disable the Sim's universe, but the trajectory paths will be retained.

`trajectory`

The trajectory file for this Sim's universe.

To change the trajectory file(s) for this Sim's Universe, set the path to the file. A single path will use a single trajectory file, while a list or tuple of paths will use each trajectory file in order for building the universe. `None` indicates that the Universe will not load a trajectory file at all (but the topology may have coordinates).

AtomSelections The class *mdsynthesis.limbs.AtomSelections* is the interface used by Sims to get `MDAnalysis.AtomGroup` objects from stored selection definitions.

class `mdsynthesis.limbs.`**`AtomSelections`**(*treant*)

Stored atom selections for the universe.

Useful atom selections can be stored for the universe and recalled later.

`add`(*handle*, **selection*)

Add an atom selection for the attached universe.

AtomGroups are needed to obtain useful information from raw coordinate data. It is useful to store AtomGroup selections for later use, since they can be complex and atom order may matter.

If a selection with the given *handle* already exists, it is replaced.

Parameters

- **handle** (*str*) – Name to use for the selection.
- **selection** (*str, AtomGroup*) – Selection string or AtomGroup indices; multiple selections may be given and their order will be preserved, which is useful for e.g. structural alignments.

`create`(*handle*)

Generate AtomGroup from universe from the given named selection.

If named selection doesn't exist, `KeyError` raised.

Parameters **`handle`** (*`str`*) – Name of selection to return as an AtomGroup.

Returns The named selection as an AtomGroup of the universe.

Return type AtomGroup

`get`(*handle*)

Get selection definition for given handle.

If named selection doesn't exist, `KeyError` raised.

Parameters **`handle`** (*`str`*) – Name of selection to get definition of.

Returns **definition** – list of strings defining the atom selection

Return type list

`keys`()

Return a list of all selection handles.

`remove`(**handle*)

Remove an atom selection for the universe.

If named selection doesn't exist, `KeyError` raised.

Parameters **`handle`** (*`str`*) – Name of selection(s) to remove.